版权相关注意事项：
1、书籍版权归善者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负。
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不错，人员，质量低下，你也可以选择退货澳斯客本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

O'REILLY
Vue.js快跑
构建触手可及的高性能Web应用
Vue.js: Up & Running
[美] Callum Macrae 著
宋鹏 刘腾辉 张海雷 吴浩亮 译
电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

译者序

在掘金问这上的一篇名为《我为什么要立刻放弃React而使用Vue？》的评文中，评论者有人说道，“新式过一样很箱，10个人里有9个使用的Vue，基础一问三不知，前使用React的人气没有，而认为，基础指行去个人，这当然可以归功于React，但却没有指出指导Vue，相反，这种现象恰恰说明了Vue的优势——上手很快，基础，这无疑是我们在这个快速迭代的时代里，选择Vue的一个潜在原因，它让我们快速地创造出许多一个可用后期的快感，并很意这份快感，进一步深入，最终选代出一个满意的应用，这就像大学里学C语言，如果一开始不是教授都要让去给你的程序，而是去一个简单的类似能调入个，我们学习语言的积极性会不会更高呢？当然，前提是Vue的性能可以和React、Angular 等前端框架比较，上手快能才有意义，发行性能方面的比较，可以查阅官方文档（https://vuejs.org/v2/guide/comparison.html）。

对于现代化前端开发体系，MVVM 框架无疑已经成为其中必不可少的一环，当下主流框架中，React、Angular、Vue 三足鼎立，我们和本书的部作者一样，在经过对它们的了解和浅现后后，最终都倒向了 Vue 而本后不久也费的开发风格，如果没有前所未有的后后及设计能力，想要优雅地使用 React 会存在一些障碍，而对于 Angular 来说，对于个人开发者和中小型的后后开发对象而言“重”了，学习曲线也比较陡峭。但 Vue 对于个人开发者和中小型后开发来说，学习曲线也是最适性的，非常适合用来入门和学习 MVVM 框架中所体现的通用知识和开发理念。

说到底，JavaScript 是一门特殊的语言，自诞生二十几年来，不断被赋予更多的责任，建立为整动后期网络期的支柱之一不足为过，而后前端开发领域中，为 jQuery 的编所化开发，到模块化和组件化，从只能 in 两种“是型时代”，到支持完善的工程化时代，前

端开发的前景和世界有了飞跃式的提升，对前端开发从业者的职业素养要求也不断提高，时至今日，Angular、React 和 Vue 三大框架覆盖了前端开发的大部分场景，无论是在企业后后从到支持不断完善，为前端开发带来极大的帮助，在非全栈开发时之外，我们更关后深入掌握框架的原理以及源码，并基考如何提升的效率，才有可能在下一个跳舞中成为时代的弄潮儿。

希望这本指前能够为将来前端人的的起到作用作指点和解的作用。

最后，非常感谢电子工业出版社的编辑老师们，一起经历了一段默契的时光，在翻译工作过程中，我们经历会实现，作为一名后前开发者，自己从对 Vue 有所了解到，但也仍然对不好会在中午发现一些不符合意的地方后，这无后人大前是不心，此外，部作的内容自也通选择后前，但翻译时，有时候就像电全部，需要不断推敲用词，组织语句，好在，团队前我此都非常负责心，这对给自己很多有益的意见，并且形成了相互的足够深的共好氛围，这也使后翻译成果变得更好。

在此，感谢团队的付出，感谢所有支持和帮助本书翻译的老师和编辑们，如果在翻译方面有错误和不足的地方，恳请读者批评指正。

目录

前言

前端开发领域一直在改变。网站变得越来越丰富，交互性也越来越强，作为前端开发者，我们需要不断增加复杂的功能和使用更加强大的工具。使用 jQuery 在某个页面中更改一部分文本很简单，但我们要做的事可不止这些——比如更新页面中大量具有交互性的部分、处理复杂的状态、使用客户端路由、高效简洁地编写和组织代码——在这种情况下，使用 JavaScript 框架会让我们的工作变得更加轻松。

框架是一种 JavaScript 工具，它使开发者能够更容易地创建丰富而又具有交互性的网站。框架所包含的功能使我们能够创建一个功能完备的 Web 应用程序，包括操作复杂的数据并将其显示在页面上、处理客户端路由而不是依赖服务端、有时甚至允许我们只需访问服务器一次并完成初始下载就能构建一个完整的网站。*Vue.js* 是一款近来十分流行的 JavaScript 框架，同时它的普及性还在扩大。当时还在 Google 工作的 Evan You 在 2014 年编写并发布了 Vue.js 的第一版。在写本书时，Vue.js 在 GitHub 上已经拥有超过 75000 个星标，这使得它成为 GitHub 上受关注度排名第八位的代码仓库，同时这个数量还在迅速上涨[1]。Vue 拥有上百位合作者，它在 npm 上每天约有 40000 次的下载量。它包含在开发网站和应用程序时非常有用的功能：一种功能强大、能够创建 DOM 和监听事件的模板语法，无须操心数据变化带来相应模板更新的响应式原理以及使维护数据变得更容易的功能。

1 当我开始着手写这本书的时候，它已经拥有了 65000 个星标。当你读到这本书的时候，它肯定已经不止 75000 个星标了！

适合读者

如果你已经对 HTML 和 JavaScript 有一定了解，并希望通过学习如何使用框架来提升对它们的认识，则这本书是为你而准备的。你没有必要精通 JavaScript，但是我在代码示例中没有解释任何有关 Vue.js 功能以外的 JavaScript 代码，所以了解一些 JavaScript 的基础知识将会很有帮助。代码示例也使用 JavaScript 的最新规范 ECMAScript 2015 进行编写，因此其中会包含诸如常量、箭头函数和解构等新的语言特性。如果你对 ES2015 不熟悉，不要担心——有大量高质量的文章和资源能够帮助你[1]，同时示例代码也拥有很好的可读性。

如果你有使用 React 的经验，这本书仍然适合你，但花一些时间查看附录 B 是值得的，其中解释了 Vue.js 中涉及的一些概念，并与你在 React 中已经了解的内容做了比较。

各章简介

本书包含 7 章和两个附录。

第 1 章 Vue.js 基础

本章介绍 Vue.js 的基础知识和核心技术。阐述如何安装并将 Vue.js 引入网页，以及如何使用它在页面上展示数据。

第 2 章 Vue.js 组件

Vue.js 允许并鼓励你将代码拆分为多个可在代码库中复用的组件。这一章将详细阐述如何创建一个易于维护和理解的代码库。

第 3 章 使用 Vue 添加样式

本书中的每一部分都会涉及 HTML 和 JavaScript，但在这一章中，将介绍更多在创建网站中关于视图层的内容。将阐述如何在 Vue 中使用 CSS 来定制化网站和应用，以及使用内置的辅助函数来协助你完成这项工作。

第 4 章 render 函数和 JSX

如果你看过了很多 Vue 代码或已经阅读完入门指南，就会熟知模板语法，但除了这个，Vue 还支持自定义渲染函数，它允许你使用 JSX 语法——一种 React 用户熟悉

1 推荐 Babel 网站上的“学习 ES2015”。

的语法。在本章中还将阐述如何在 Vue 中使用 JSX 语法。

第 5 章 使用 vue-router 实现客户端路由

Vue 本身只是一个视图层。要创建一个具有多个页面且无须新的额外请求即可访问的应用程序（或者用专业的说法：单页应用），需要将 vue-router 添加到网站中，可以使用它来处理客户端路由——比如说请求指定的路径时，代码如何执行和展示数据。这一章将阐述如何做到这一点。

第 6 章 使用 vuex 实现状态管理

在一个具有多级组件层级、更加复杂的应用中，在组件中传递数据会变得有些棘手。Vuex 使你能够在一个集中的空间里处理应用的状态，本章将阐述如何使用它来轻松处理复杂应用的状态。

第 7 章 对 Vue 组件进行测试

到此，已经学习到了能够使网站正常运作所需要了解的一切，但是如果需要在未来继续维护网站，你应该为它编写测试。这一章将介绍如何使用 vue-test-utils 来为组件编写单元测试以确保它们在未来的运行中不会出现问题。

附录 A 搭建 Vue 开发环境

vue-cli 使你能够从给定的模板中，快速构建 Vue 应用，本附录向你展示它是如何工作的以及它提供的一些模板。

附录 B Vue 与 React

如果你拥有使用 React 的经验，那么可能已经很熟悉 Vue 中的诸多概念。本附录重点介绍 Vue 和 React 之间的异同点。

风格指南

本书中的所有示例代码都遵循官方 Vue 风格指南中所列举的指导原则。一旦你理解了 Vue 并希望创建一个大型应用或者与他人进行协作，推荐你去阅读风格指南并遵循其中的指导原则。

可以在 Vue.js 的网站上找到风格指南。

约定

下面是本书中所使用的排版约定：

中文楷体或英文斜体（*Italic*）

用于新的术语、URLs、电子邮件地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序清单以及段落内所涉及的程序元素，如变量或函数名称、数据库、数据类型、环境变量、语句和关键字。

等宽字体加粗（**`Constant width bold`**）

用于应当被用户输入的指令或者其他文本。

等宽字体加斜（*`Constant width italic`*）

用于应当被用户所提供的值或者取决于上下文的值所替换的文本。

这个元素代表技巧或者建议。

这个元素代表通用的备注。

这个元素代表警告或者提醒。

如何联系我们

请将对本书的评价和发现的问题通过如下地址通知出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过 *http://bit.ly/reactNativeCookbook* 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 *bookquestions@oreilly.com*。

更多有关书籍、课程、会议和新闻的信息，请见网站 *http://www.oreilly.com*。

在 Facebook 找到我们：*http://facebook.com/oreilly*。
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*。
在 YouTube 上观看：*http://www.youtube.com/oreillymedia*。

致谢

首先，我向所有在我编写本书时，被我冷落的朋友们，表示抱歉和感谢。现在我已经写完了，我们又可以愉快地交流了。

特别感谢 Michelle，因为他一直都很棒，并且对音乐有很好的品位。

非常感谢 JuhoVepsäläinen 和 Rob Pemberton，他们帮助我处理 React 示例。我使用 React 已有一段时间，所以我很感谢他们的帮助！还要感谢其他人激发了我在写作时脑海中跳过的想法、句子和文字：Sab、Ash、Alex、Chris、Gaffen 和 Dave 等。

感谢 O'Reilly Media 的每一位，并特别感谢 Chris Fritz、Jakub Juszczak、Kostas Maniatis 和 Juan Vega，他们为本书提供了很多技术反馈，我从中也学到了很多。

读者服务

轻松注册成为博文视点社区用户（*www.broadview.com.cn*），扫码直达本书页面。

- **下载资源**：本书提供示例代码及资源文件，可在 **下载资源** 处下载。
- **提交勘误**：您对书中内容的修改意见可在 **提交勘误** 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 **读者评论** 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/35299*

第 1 章

Vue.js基础

如前言所述，*Vue.js* 是构建强大客户端应用的技术体系中的核心库。但开发一个网站并非必须使用一整套技术方案，因此我们会从 Vue 自身开始讲起。

为什么选择 Vue.js

如果不使用框架，项目最终会成为一团不可维护的代码，而绝大部分代码所要处理的工作，框架都已经为我们抽象出来了。以下面两段代码为例，它们都实现了同样的功能：使用 Ajax 下载一个列表数据并在页面上显示。其中，第一段使用 jQuery 实现，而第二段使用 Vue 实现。

使用 jQuery 的代码逻辑是下载列表数据，选择 `ul` 元素，然后判断列表项是否存在，如果存在则遍历列表，手动创建 `li` 元素，根据需要添加 `is-blue` 类名，把列表项设置为 `li` 的文本子节点，最后把生成的 `li` 节点添加到 `ul` 元素中：

```
<ul class="js-items"></ul>

<script>
  $(function () {
     $.get('https://example.com/items.json')
       .then(function (data) {
         ar $itemsUl = $('.js-items');

         if (!data.items.length) {
           var $noItems = $('li');
           $noItems.text('Sorry, there are no items.');
```

```
        $itemsUl.append($noItems);
      } else {
        data.items.forEach(function (item) {
          var $newItem = $('li');
          $newItem.text(item);

          if (item.includes('blue')) {
            $newItem.addClass('is-blue');
          }

          $itemsUl.append($newItem);
        });
      }
    });
  });
</script>
```

这段代码执行了以下逻辑：

1. 使用 `$.get()` 发起 Ajax 请求。
2. 选取类名为 `.js-items` 的元素并将它存入 `$itemUl` 变量。
3. 如果下载的列表中没有列表项，则创建一个 `li` 元素，设置这个 `li` 元素的文本节点以表示当前没有列表项，然后将它添加到文档流中。

   如果列表中存在列表项，则循环遍历每一个列表项。
4. 为列表中的每一项创建一个 `li` 元素，并将其文本节点设置为该列表项的值。然后判断该列表项是否包含 `blue` 字符串，如果包含则将 `li` 元素的类名设置为 `is-blue`。最后，将 `li` 元素添加到文档流中。

上面的每一步都不得不手动实现——每一个元素都需要单独创建并添加到文档流中。而且代码逻辑乍看起来并不清晰，我们不得不阅读所有的代码来弄懂发生了什么。

使用 Vue 实现同样功能的代码则更加容易阅读和理解——即使你还不熟悉 Vue：

```
<ul class="js-items">
  <li v-if="!items.length">Sorry, there are no items.</li>
  <li v-for="item in items" :class="{ 'is-blue': item.includes('blue') }">
    {{ item }}</li>
</ul>
```


```
<script>
  new Vue({
    el: '.js-items',
    data: {
      items: []
    },
    created() {
      fetch('https://example.com/items.json')
        .then((res) => res.json())
        .then((data) => {
          this.items = data.items;
        });
    }
  });
</script>
```

这段代码执行了以下逻辑：

1. 使用 fetch() 发起一个 Ajax 请求。
2. 将返回的 JSON 数据解析为 JavaScript 对象。
3. 将下载的列表项存储为 data 对象中的 items 属性。

上述便是这段代码的所有逻辑。现在已经下载并存储了列表项，然后可以使用 Vue 的模板功能将这些元素渲染到文档对象模型（Document Object Model，简称 DOM，用于标明元素如何在 HTML 页面上呈现）。我们只需要告诉 Vue 我们希望为每个列表项创建一个 li 元素，并且将 item 作为该元素的文本，Vue 会为我们处理元素的创建和类名的设置。

如果还不能完全理解这段示例代码，请不要担心，放慢节奏，通过整本书逐一介绍各种各样的概念。

第二段代码明显更短，而且更加易读，这是因为实际的应用逻辑和视图逻辑是完全隔离的。和不得不艰难地查看一些 JQuery 代码来了解当时添加了哪些元素不同，我们可以看一下上面的 Vue 模板：如果列表项没有展示一段警告信息，则列表项会用列表元素显示。在大型示例中，它们的差别会变得更加明显。想象一下，要在页面上增加一个按钮，当用户单击这个按钮时向服务器发送请求，获取新的列表项，然后更新页面。要实现这个功能，在 Vue 的例子中只需要增加几行额外的代码，但在 jQuery 的例子中，事情会变得愈加复杂。

除了核心的 Vue 框架，还有几个库和 Vue 配合起来非常好用，而且是由 Vue 的原班人马来维护的。vue-router 用于路由控制——根据应用的不同 URL 来显示不同的内容。vuex 用于状态管理——通过一个全局数据中心在组件间共享数据。vue-test-utils 用于 Vue 组件的单元测试。本书后续内容将对这 3 个库进行恰当的介绍：第 5 章介绍 vue-router，第 6 章介绍 vuex，第 7 章介绍 vue-test-utils。

安装和设置

安装 Vue 不需要任何特殊的工具，使用下面的代码就可以实现：

```
<div id="app"></div>
<script src="https://unpkg.com/vue"></script>
<script>
  new Vue({
    el: '#app',
    created() {
      // 这段代码会在应用启动时运行
    }
  });
</script>
```

这段代码有 3 个重点。首先有一个 ID 为 app 的 div 元素用于初始化 Vue ——因为多种原因，不能在 body 元素上进行初始化。然后，在页面上引用 CDN[1] 版本的 Vue 文件。当然也可以下载到本地并引用，但出于简化的考虑，暂且这样处理。最后，运行一些 JavaScript 代码，创建一个 Vue 的实例，并将该实例的 el 属性指向之前提到的 div 元素。

上述方式对于简单的页面很好用，但对于任何更加复杂的场景，你可能希望使用类似 webpack 这样的打包工具。借助它可以使用 ECMAScript 2015（甚至更高）标准的 JavaScript 语言、编写单文件组件、实现组件的相互引用，以及书写作用域为特定组件的 CSS（在第 2 章中会有详细介绍）等其他特性。

vue-loader 和 webpack

vue-loader 是一个 webpack 的加载器，允许你将一个组件的所有 HTML、JavaScript 和 CSS 代码编写到同一个文件中。目前只需要了解如何安装它，我们将会在第 2 章中适当

1 内容分发网络（CDN）是将资源托管到全世界各处的服务器上以实现快速分发。CDN 版本对于开发和快速验证比较有用，但在将 unpkg 应用于生产环境前，需要检查它是否适合你。

深入地探索它。如果已经配置好 webpack 或者有喜爱的 webpack 配置模板，可以通过 npm 安装 vue-loader，然后将下面的代码添加到 webpack 配置的 loader 部分，就可以完成 vue-loader 的安装。

```
module: {
  loaders: [
    {
      test: /\.vue$/,
      loader: 'vue',
    },
    // ... 其他加载器 ...
  ]
}
```

如果还没有配置好 webpack 或者正在头疼如何添加 vue-loader，不要担心，我也没有从无到有地完成过 webpack 的配置。可以使用一份现成的模板来初始化 vue 项目，它使用了 webpack 并已经安装好了 vue-loader。你可以通过 vue-cli 使用它：


```
$ npm install --global vue-cli
$ vue init webpack
```

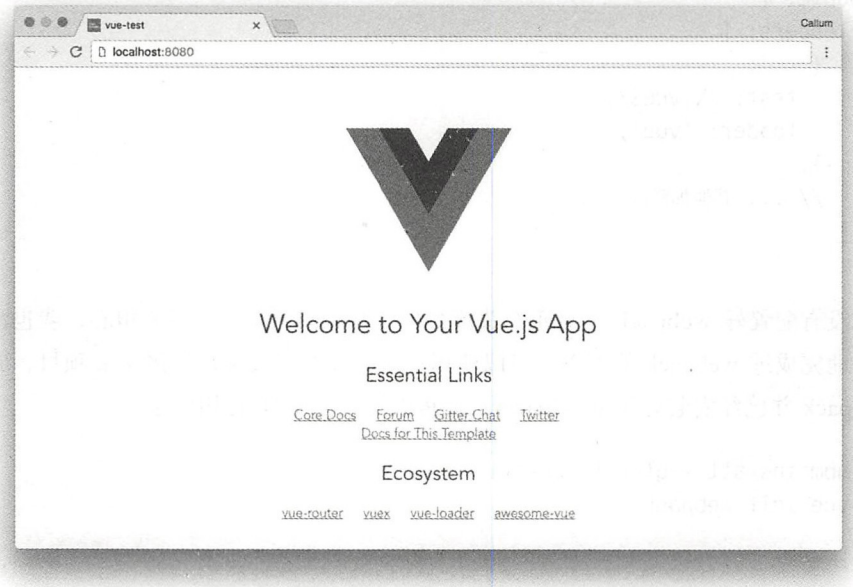
执行上述命令后你会被问到一系列的问题，例如项目的名称以及它需要哪些依赖，一旦回答完这些问题，vue-cli 会为你初始化一个基础的项目：

```
> vue init webpack

? Generate project in current directory? Yes
? Project name vue-test
? Project description A Vue.js project
? Author Callum Macrae <callum@macr.ae>
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? No
? Setup unit tests with Karma + Mocha? No
? Setup e2e tests with Nightwatch? No

   vue-cli · Generated "vue-test".

   To get started:

     npm install
     npm run dev

   Documentation can be found at https://vuejs-templates.github.io/webpack
```


现在就尝试一下，然后根据命令行工具的提示来启动服务器。

恭喜——你已经创建了你的第一个 Vue 工程！

6



可以查看生成的文件来了解发生了什么，大部分重要的内容在 *src* 目录下的 *.vue* 文件中。

模板（Template）、数据（Data）和指令（Directive）

Vue 的核心是将数据显示在页面上，这一功能通过模板实现。为正常的 HTML 添加特殊的属性——被称作指令——借助它来告诉 Vue 我们想要实现的效果以及如何处理提供给它的数据。

来看一个例子。下面的示例会在早上显示“早上好！”，在下午 6 点前显示“中午好！”，在之后显示“晚上好！”：

```
<div id="app">
  <p v-if="isMorning">早上好 !</p>
  <p v-if="isAfternoon">中午好 !</p>
  <p v-if="isEvening">晚上好 !</p>
```


```
</div>
<script>
   var hours = new Date().getHours();

   new Vue({
     el: '#app',
     data: {
       isMorning: hours < 12,
       isAfternoon: hours >= 12 && hours < 18,
       isEvening: hours >= 18
     }
   });
 </script>
```


首先来看最后一部分：data 对象，我们通过它告诉 Vue 想在 template 上显示哪些内容。在这个对象上设置了 3 个属性——isMorning、isAfternoon 和 isEvening——它们中有一个值为 true，其他两个值为 false，具体根据当前时间决定。

之后，在 template 中使用 v-if 指令来根据 3 个变量的值显示 3 种问候语中的一种。设有 v-if 属性的元素只有传递给指令的值为真时才会显示，否则，该元素不会被写入页面。如果当前时间为下午 2:30，那么如下内容会被输出到页面上：

```
<div id="app">
  <p>中午好！</p>
</div>
```

尽管 Vue 具备响应式能力，但上述例子并非响应式的，页面也不会随着时间变化而更新。稍后我们会更详细地介绍响应式。

然而上述示例中出现了很多重复代码：如果能够将当前时间设置为 data 的一个属性，然后在 template 中实现比较逻辑，这样代码将会更加简洁。幸运的是我们可以这样做，Vue 支持在 v-if 中进行简单的表达式运算：

```
<div id="app">
  <p v-if="hours < 12">早上好！</p>
  <p v-if="hours >= 12 && hours < 18">中午好！</p>
  <p v-if="hours >= 18">晚上好！</p>
```


```
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      hours: new Date().getHours()
    }
  });
</script>
```

在 JavaScript 中实现业务逻辑，而在 template 中实现视图逻辑，以这种方式编写代码意味着我们可以对页面上将会显示的内容一目了然。相比让离元素很远的 JavaScript 代码来决定该元素是显示还是隐藏，这种方法要好很多。

8 后面还会介绍计算属性，它可以使上面的代码更加简洁——上述代码仍然有些重复。

除了使用指令，也可以通过插值的方式将数据传递给模板，如下所示：

```
<div id="app">
  <p>Hello, {{ greetee }}!</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      greetee: 'world'
    }
  });
</script>
```

上述代码将会输出如下内容到页面上：

```
<div id="app">
  <p>Hello, world!</p>
</div>
```

可以将上述两者结合起来，同时使用指令和插值来只显示一些被定义的或者有用的文本。来试一下，看你是否能指出下面的代码在什么情况下会显示什么内容到页面上？

```
<div id="app">
  <p v-if="path === '/'">你正位于首页</p>
  <p v-else>你正位于 {{ path }}</p>
```


```
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      path: location.pathname
    }
  });
</script>
```

location.pathname 是当前页面 URL 的路径值，因此对于网站的首页它的值会是“/”，对于网站的其他页面，它的值就会是其他值（比如可能是“/post/1635”）。上述示例代码会告诉你当前页面是否是网站的首页：它在 v-if 指令中检测路径值是否等于“/”（由此可以判断用户是否位于网站的首页），然后会遇到一个新指令：v-else。这个指令很简单，当用在带有 v-if 指令的元素之后时，它的表现和一个 if-else 语句中的 else 语句一样。所以第二个元素只在第一个元素不显示时才会显示在页面上。

正如你所见，除了可以传递字符串和数字，还可以传递其他类型的数据到模板中。因为可以在模板中执行简单的表达式运算，所以可以传递一个数组或对象，并在模板中引用它的某个属性或元素：


```
<div id="app">
  <p>第二条狗的名字是 {{ dogs[1] }}</p>
  <p>所有狗的名字是 {{ dogs }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      dogs: ['Rex', 'Rover', 'Henrietta', 'Alan']
    }
  });
</script>
```

上述代码将会输出如下内容到页面上：

```
<p>第二条狗的名字是 Rover</p>

<p>所有狗的名字是 [ “Rex", “Rover", “henrietta", “Alan" ]</p>
```


可见，如果将整个数组或对象输出到页面上，Vue 会输出 JSON 编码后的值。在调试时，这样做比将日志输出到控制台更加有效，因为页面显示会随着值的变化而更新。

v-if vs v-show

在前面一节中知道了 v-if 指令可以控制一个元素的显示和隐藏，那么它是如何实现的？它和看起来很像的 v-show 指令有什么区别呢？

如果 v-if 指令的值为假[1]，那么这个元素不会被插入 DOM 。

下面的 Vue 模板

```
<div v-if="true">one</div>
<div v-if="false">two</div>
```

会输出如下内容 ：

```
<div>one</div>
```

和 v-show 指令比较一下，该指令使用 CSS 样式控制元素的显示 / 隐藏。

10 下面的 Vue 模板

```
<div v-show="true">one</div>
<div v-show="false">two</div>
```

会输出如下内容 ：

```
<div>one</div>
<div style="display: none">one</div>
```

你的用户（可能）会看到同样的内容，但两者间确实存在其他的影响和差异。

首先，因为使用 v-if 隐藏的内部元素不会被显示，Vue 不会尝试生成对应的 HTML 代码；而对于 v-show 指令，事实并非如此。这意味着隐藏尚未加载的内容时，v-if 指令更好一些。

下面这个例子会抛出一个异常 ：

```
<div id="app">
```

1 假值包括 false、undefined、null、"" 和 NaN

```
    <div v-show="user">
      <p>User name: {{ user.name }}</p>
    </div>
  </div>

  <script>
    new Vue({
      el: '#app',
      data: {
        user: undefined
      }
    });
  </script>
```

会抛出异常是因为 Vue 尝试执行 user.name——一个尚不存在的对象的属性。在该示例中如果使用 v-if 就会正常工作，因为 Vue 直到 v-if 语句为真时才会去尝试生成元素的内部内容。

此外，有两个条件语句和 v-if 有关：v-else-if 和 v-else。它们的行为非常符合预期：

```
<div v-if="state === 'loading'">加载中 ...</div>
<div v-else-if="state === 'error'">出错了</div>
<div v-else>... 我们的内容！ </div>
```

当 state 值为 loading 时会显示第一个 div，state 值为 error 时会显示第二个，state 为其他任意值时会显示第三个 div。同一时间只有一个元素会显示。

那么，了解了这么多，为什么会有人想要使用 v-show 呢？

使用 v-if 会有性能开销。每次插入或者移除元素时都必须要生成元素内部的 DOM 树，这在某些时候是非常大的工作量。而 v-show 除了在初始创建开销时之外没有额外的开销。如果希望频繁地切换某些内容，那么 v-show 会是最好的选择。

此外，如果元素包含任何图片，那么仅使用 CSS 隐藏父节点可以使浏览器在图片显示之前就加载它，这意味着一旦 v-show 变为真值，图片就可以显示出来。如果是 v-if 指令，图片会直到要显示时才开始加载。

模板中的循环

另外一个我经常使用的指令是 v-for，这个指令通过遍历一个数组或者对象，将指令所

在的元素循环输出到页面上。请看下面示例：

```
<div id="app">
  <ul>
  <li v-for="dog in dogs">{{ dog }}</li>
  </ul>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      dogs: ['Rex', 'Rover', 'Henrietta', 'Alan']
    }
  });
</script>
```

上述代码将数组中的每一项输出到页面的列表元素中，具体如下所示：

```
<div id="app">
  <ul>
    <li>Rex</li>
    <li>Rover</li>
    <li>Henrietta</li>
    <li>Alan</li>
  </ul>
</div>
```

v-for 指令同样支持对象的遍历。下面这个例子将一个包含部分城市平均租金的对象输出到页面上：

```
<div id="app">
  <ul>
    <li v-for="(rent, city) in averageRent">
      {{ city }}的平均租金是 ${{ rent }}
    </li>
  </ul>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      averageRent: {
```


```
        london: 1650,
        paris: 1730,
        NYC: 3680
      }
    }
  });
</script>
```

这里语法稍微有些不同，因为我们还希望能够获取对象的键名——只在页面上显示租金但不显示城市的名称并没有太大的意义。但是，如果不需要键名，那么仍然可以使用和前面相同的语法：`v-for="rent in averageRent"`。上述逻辑对数组同样有效：如果想获取数组的索引，可以对数组使用你刚见过的括号和逗号语法：`v-for="(dog, i) in dogs"`。

请记住参数的顺序是：`(value, key)`。

最后，如果只想要个简单的计数器，可以传一个数字作为参数。下面这个示例会依次输出 1 到 10：

```
<div id="app">
  <ul>
    <li v-for="n in 10">{{ n }}</li>
  </ul>
</div>
<script>
  new Vue({
    el: '#app'
  });
</script>
```

你可能期待输出 0 到 9，但事实并非如此。如果希望列表从 0 开始，一个简单的办法是使用 `n-1` 来代替 `n`。

属性绑定

有些指令，例如 `v-bind`，可以接收参数。`v-bind` 指令用于将一个值绑定到一个 HTML 属性上。例如，下面的这个例子将 `submit` 值绑定到按钮的 `type` 属性上：

```
<div id="app">
  <button v-bind:type="buttonType">Test button</button>
```


```
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      buttonType: 'submit'
    }
  });
</script>
```

输出到文档中的内容如下所示：

```
<button type="submit">Test button</button>
```

v-bind 是指令的名称，type 是指令的参数：在这个示例中即为属性的名称，我们想要将给定的变量绑定到该属性上。buttonType 即是给定的变量。

这个指令对 disabled 和 checked 之类的属性同样有效：如果传入的表达式为真，则输出的元素会带有这个属性；如果为假，元素则不会带有这个属性：

```
<div id="app">
  <button v-bind:disabled="buttonDisabled">Test button</button>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      buttonDisabled: true
    }
  });
</script>
```

当在大量的属性上使用 v-bind 指令时，反复书写指令名称是相当重复的工作。一种简写方式是可以省略 v-bind 部分，而用冒号代替。下面的例子使用这种简写语法重写了前面的例子：

```
<div id="app">
  <button :disabled="buttonDisabled">Test button</button>
</div>
<script>
  new Vue({
    el: '#app',
```


```
    data: {
      buttonDisabled: true
    }
  });
</script>
```

尽管选择使用哪种语法是个人喜好，但我更倾向简写语法而且很少在代码中写 v-bind。

无论选择使用 v-bind 还是简写语法，尽量保持一致。在某些地方使用 v-bind，而在另外的地方又使用简写语法，这样会让代码变得有些凌乱。

响应式

本章上面几节已经展示了如何使用 Vue 将 JavaScript 中的值输出到 HTML 中并转化为 DOM——但这又如何？是什么令 Vue 与其他模板语言相比显得与众不同？

除了在一开始创建 HTML，Vue 还监控 data 对象的变化，并在数据变化时更新 DOM。为了论证这一点，让我们创建一个用于显示页面已经打开多久的计时程序。只需要一个变量，称为 seconds，然后使用 setInterval 每隔一秒将这个变量递增 1。

```
<div id="app">
  <p>自从你打开这个页面，已经过了 {{ seconds }} 秒。</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      seconds: 0
    },
    created() {
      setInterval(() => {
        this.seconds++;
      }, 1000);
    }
  });
</script>
```


created 函数方法会在应用初始化完成后执行。Vue 的生命周期钩子会在后面详细介绍，所以暂时不用太过担心这个函数。这个方法中的 this.seconds 直接指向 data 对象中的相应值，并且通过操控它来更新模板。

下面就是这个页面打开一会儿之后的一个示例效果：

自从你打开这个页面，已经过了 14 秒。

Vue 的响应式能力除了在使用插值将数据输出到页面上时有效，在将 data 对象的属性作为指令值时也同样有效。举一个例子，如果我们在上一节 v-bind 的示例中每秒执行一次 this.buttonDisabled = !this.buttonDisabled;，可以看到按钮的 disabled 属性会每秒切换一次：按钮会在某一秒内可用，然后在下一秒内被禁用。

15

响应式是令 Vue 如此强大的一个极其重要的特性和部分。你会非常频繁地在本书和其他任何你做的 Vue 项目中看到它。

响应式如何实现

这部分内容有一些超前，如果无法弄懂可以先跳过，待读完后续章节再回过头来阅读这一部分。

Vue 修改了每个添加到 data 上的对象，当该对象发生变化时 Vue 会收到通知，从而实现响应式。对象的每个属性会被替换为 getter 和 setter 方法，因此可以像使用正常对象一样使用它，但当你修改这个属性时，Vue 会知道它发生了变化。

以下面这个对象为例：

```
const data = {
  userId: 10
};
```

当 userId 发生变化时，你如何得知它发生变化了呢？可以存储这个对象的一个副本，然后比较二者，但这并不是最高效的方法。这种方法称为 脏检查，也是 Angular1 所采用的方法。

另外一种方法是，使用 Object.defineProperty() 覆写这个属性：

```
const storedData = {};
```


```
  storedData.userId = data.userId;

  Object.defineProperty(data, 'userId', {
    get() {
      return storedData.userId;
    },
    set(value) {
      console.log('User ID changed!');
      storedData.userId = value;
    },
    configurable: true,
    enumerable: true
  };
```

上述代码和 Vue 的实现并不完全一致，但它很好地展现了 Vue 的思路。

同时 Vue 还使用代理方法封装了被观察的数组对象上的一些数组方法（例如 `.splice()`），来观察数组方法何时被调用。这也是为什么当你调用 `.splice()` 时，Vue 知道你更新了数组，并触发了必要的视图更新。

如果想了解更多关于 Vue 响应式实现的工作原理，请查阅官方文档“深入响应式原理”。 16

注意事项

响应式的使用存在一些限制。理解 Vue 的响应式原理可以帮助你弄明白这些限制，你也可以只是简单地直接记住这些限制。因为只存在少数几条限制，所以记住它们并不困难。

为对象添加新的属性

因为 getter/setter 方法是在 Vue 实例初始化的时候添加的，只有已经存在的属性是响应式的；当为对象添加一个新的属性时，直接添加并不会使这个属性成为响应式的：

```
const vm = new Vue({
  data: {
    formData: {
      username: 'someuser'
    }
  }
});
```


```
vm.formData.name = 'Some User';
```

尽管 formData.username 属性是响应式的，并且会对变化做出响应，但 formData.name 属性并非如此。有几种方法处理这种情况。

最简单的办法是在初始化时在对象上定义这个属性，并把它的值设置为 undefined。上例中的 formData 对象会变成下面这样：

```
formData: {
  username: 'someuser',
  name: undefined
}
```

或者，也可以使用 Object.assign() 来创建一个新的对象然后覆盖原有对象，当需要一次性更新多个属性时，这是最有效的办法：

```
vm.formData = Object.assign({}, vm.formData, {
  name: 'Some User'
});
```

最后，Vue 还提供了 Vue.set() 方法，可以使用它将属性设置为响应式的：

```
Vue.set(vm.formData, 'name', 'Some User');
```

在组件内部也可以使用 this.$set 来调用这个方法。

17 设置数组的元素

不能直接使用索引来设置数组的元素。下面的做法是行不通的：

```
const vm = new Vue({
  data: {
    dogs: ['Rex', 'Rover', 'Henrietta', 'Alan']
  }
});

vm.dogs[2] = 'Bob'
```

有两种方法可以处理这一问题。一种是使用 .splice() 方法移除旧元素并添加新元素：

```
vm.dogs.splice(2, 1, 'Bob');
```

另一种是使用 Vue.set():

```
Vue.set(vm.dogs, 2, 'Bob');
```

两种方法的效果是一样的。

设置数组的长度

在 JavaScript 中，可以设置一个数组的长度，自动让空元素填充数组至该长度或者截掉数组的尾部（具体根据设置的长度比原长度长还是短来决定）。不过这个方法不能用于处理 data 对象中的数组，因为 Vue 不能检测到该操作对数组的任何更改。

可以用 splice 代替：

```
vm.dogs.splice(newLength);
```

这一方法只能用于缩短数组，并不能扩展它的长度。

双向数据绑定

到目前为止已经了解了如何将数据输出到模板，以及 Vue 的响应式能力如何做到让模板随着数据的更新而更新。但这只是单向数据绑定。试着运行下面的代码，你会发现 inputText 会保持不变，输入框下面的文本也会保持不变：

```
<div id="app">
  <input type="text" v-bind:value="inputText">
  <p>inputText: {{ inputText }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      inputText: 'initial value'
    }
  });
</script>
```

18

如果想让上面这个例子按预期运行，就不能使用 v-bind:value——使用 v-bind 只会在 inputText 值变化时才更新输入框的值，反过来则不行。作为替代，可以使用 v-model 指令，它作用于输入框元素，将输入框的值绑定到 data 对象的对应属性上，因此输入框不但会接收 data 上的初始值，而且当输入内容更新时，data 上的属性值也会更新。将上

面那个例子的 HTML 部分替换为下面的代码，现在当改变输入框的内容时，inputText: initial value 的值就会跟着更新：

```
<div id="app">
  <input type="text" v-model="inputText">
  <p>inputText: {{ inputText }}</p>
</div>
```

使用 v-model 时一定要记住，如果设置了 value、checked 和 selected 属性，这些属性会被忽略。如果想设置输入元素的初始值，应该在 data 对象中设置。来看一下下面这个例子：

```
<div id="app">
  <input type="text" v-bind:value="inputText" value="initial value" />
  <p>inputText: {{ inputText }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      inputText: ''
    }
  });
</script>
```

在上面这个例子中，inputText 仍然会绑定到 inputText 变量，输入到输入框中的任何内容都会显示在下面的段落标签中。但是，这个输入框没有初始值，因为它没有像前一个例子那样在 data 对象中设置初始值，而是使用了 input 元素的 value 属性。

至于输入元素 (input)、多行文本框 (textareas)、下拉列表和复选框，这些元素基本上都不会有什么意外：输入的值和 data 对象中的值都能保持一致（对于复选框，data 对象中的值是一个布尔值）。单选框有一点不同，因为同一个 v-model 会对应多个不同的元素，其 name 属性也会被忽略，存储在 data 中的值等于当前选中的单选输入框的 value 属性的值：

```
<div id="app">
  <label><input type="radio" v-model="value" value="一"> 一</label>
  <label><input type="radio" v-model="value" value="二"> 二</label>
  <label><input type="radio" v-model="value" value="三"> 三</label>
```


```
    <p>选中的 value 值为 {{ value }}</p>
  </div>
  <script>
    new Vue({
      el: '#app',
      data: {
        value: '一'
      }
    });
  </script>
```

当第一个选择框选中时，value 的值是 一；当第二个选择框选中时，value 的值是 二，以此类推。尽管可以继续使用 name 属性，但 Vue 会忽略它，况且单选框没有这个属性也可以正常运行（同一时间只有一个单选框被选中）。

动态设置 HTML

有时候你可能希望通过一个表达式设置某个元素的 HTML。比如说调用一个 API，这个 API 返回一些需要显示在页面上的 HTML。在理想情况下，这个 API 会返回一个 JSON 对象，可以将它传给 Vue 并自行处理模板，但问题来了。Vue 内建了自动 HTML 转义功能，因此当你尝试插入 {{ yourHtml }} 时，其中的 HTML 字符会被转义——类似于 this——而且 API 返回的 HTML 将会直接以文本的方式显示在页面上。这一点都不理想！

如果想将 HTML 渲染到页面上，可以像下面这样使用 v-html 指令：

```
<div v-html="yourHtml"></div>
```

这样，无论 yourHtml 中包含什么 HTML 内容，都不会被转义而是直接输出到页面上。

务必慎用这一功能！从某个变量中取出 HTML 并输出到页面上，你可能将自己暴露在 XSS[1] 风险中。永远不要将用户输入或者允许用户修改的内容置于 v-html 中，除非对他们输入的内容提前进行了仔细的校验和转义。不然你可能会一不小心允许了用户在你的网站上执行恶意脚本。记住，只用 v-html 处理你信任的数据。

1 跨站脚本攻击 (XSS) 允许其他人在你的网站上运行任何代码。

20 方法

这一部分讲解如何使用 Vue 的方法，从而可以在模板里调用函数来处理数据。

函数是十分优雅的语言特性，它让我们可以采用可复用的方式存储一段逻辑，从而不用重复代码就可以多次使用这一逻辑。在 Vue 的模板里，函数被定义为方法来使用。正如下面这个例子所示，只要将一个函数存储为 methods 对象的一个属性，就可以在模板中使用它：

```
<div id="app">
  <p>当前状态：{{ statusFromId(status) }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      status: 2
    },
    methods: {
      statusFromId(id) {
        const status = ({
          0: '睡觉',
          1: '吃饭',
          2: '学习 Vue'
        })[id];
        return status || '未知状态：' + id;
      }
    }
  });
</script>
```

上面这个例子将代表状态的数字——可能在另外的步骤中由某个 API 提供——转换为可读的、描述真实状态的字符串。可以通过将函数设置为 methods 对象的属性来添加方法。

除了在插值中使用方法，还可以在属性绑定中使用它们——实际上，任何可以使用 JavaScript 表达式的地方都可以使用方法。下面是一个简单的例子：

```
<div id="app">
  <ul>
    <li v-for="number in filterPositive(numbers)">{{ number }}</li>
```


```
  </ul>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      numbers: [-5, 0, 2, -1, 1, 0.5]
    },
    methods: {
      filterPositive(numbers) {
        return numbers.filter((number) => number >= 0);
      }
    }
  });
</script>
```


上面这个例子中的方法接收一个数组，然后返回一个过滤掉所有负数的新数组，因此页面上会输出一个正数列表。

至于前面介绍的响应式，你会很高兴地发现它在这个例子里同样有效。如果 numbers 发生变化——例如增加或移除一个数字——这个方法会被重新调用来处理新的数字，从而输出到页面上的信息也会随之更新。

this

在方法中，this 指向该方法所属的组件。可以使用 this 访问 data 对象的属性和其他方法：

```
<div id="app">
  <p>所有正数的总和是 {{ getPositiveNumbersSum() }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      numbers: [-5, 0, 2, -1, 1, 0.5]
    },
    methods: {
      getPositiveNumbers() {
        // 注意我们使用的是 this.numbers
```


```
        // 它直接指向 data 对象的 numbers 数组
        return this.numbers.filter((number) => number >= 0);
      },
      getPositiveNumbersSum() {
        return this.getPositiveNumbers().reduce((sum, val) => sum + val);
      }
    }
  });
</script>
```

在这个例子中，getPositiveNumbers 方法中的 this.numbers 指向 data 对象中的 numbers 数组，在前面的例子中我们以参数的方式传递这个数组；而 this.getPositiveNumbers() 则指向 methods 对象中被命名为该名称的方法。

在后面的章节中会看到 this 还可以访问其他的东西。

22 计算属性

计算属性介于 data 对象的属性和方法两者之间：可以像访问 data 对象的属性那样访问它，但需要以函数的方式定义它。

请看下面这个示例，这个例子将存储在 numbers 数组中的数字加在一起，然后将总和输出到页面上：

```
<div id="app">
  <p>数字的总和是：{{ numberTotal }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      numbers: [5, 8, 3]
    },
    computed: {
      numberTotal() {
        return numbers.reduce((sum, val) => sum + val);
      }
    }
  });
</script>
```


尽管可以将计算数字总和的语句整个写在模板中，但将其单独定义在别处明显更加方便和可读。同样，不论是在方法中，还是在其他计算属性中，抑或是这个组件的任意地方，都可以通过 this 来访问到这个计算属性。和 Vue 方法一样，当 numbers 发生变化时，numberTotal 也会同时变化，并且这种变化会体现在模板上。

但是除了明显的语法区别，使用计算属性和使用方法之间有什么区别呢？确实，它们之间存在几点不同。

首先，计算属性会被缓存：如果在模板中多次调用一个方法，方法中的代码在每一次调用时都会执行一遍；但如果计算属性被多次调用，其中的代码只会执行一次，之后的每次调用都会使用被缓存的值。只有当计算属性的依赖发生变化时，代码才会被再次执行：例如，在上面这个例子中，如果向 numbers 中添加一项，numberTotal 中的代码会再次执行以获取新的计算值。因为这种方式可以确保代码只在必要的时刻执行，所以适合处理一些潜在的资源密集型工作。

你可以通过向一个被反复调用的计算属性添加 console.log() 语句来观察这种缓存特性。你会注意到这个 console.log() 语句，乃至整个计算属性，都只有在 value 变化时才会被执行：


```
<script>
  new Vue({
    el: '#app',
    data: () => ({
      value: 10,
    }),
    computed: {
      doubleValue() {
        console.log('doubleValue computed property changed');

        return this.value * 2;
      }
    }
  });
</script>
```


在上面这个例子中，只有当应用初始化和每次 value 发生变化时，字符串才会被输出到控制台。

计算属性和方法的另外一个区别是，除了能像上例展示的那样获取计算属性的值，还可以设置计算属性的值，并且在设置过程中做一些操作。实现这一点需要将计算属性由函数改为带有 get 和 set 属性的对象。例如上面这个例子，假设我们想要添加一个新功能，通过对 numberTotal 加上或减去某个值来将该值添加到 numbers 数组中。具体实现上，可以增加一个 setter 方法来比较计算属性的新值和旧值，并将两者的差值添加到 numbers 数组的末尾：

```
<div id="app">
  <p>Sum of numbers: {{ numberTotal }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      numbers: [5, 8, 3]
    },
    computed: {
      numberTotal: {
        get() {
          return numbers.reduce((sum, val) => sum + val);
        },
        set(newValue) {
          const oldValue = this.numberTotal;
          const difference = newValue - oldValue;
          this.numbers.push(difference);
        }
      }
    }
  });
</script>
```

24 此时，在组件的其他任意位置执行 this.numberTotal += 5 就可以将数字 5 添加到 numbers 数组的末尾——真是太巧妙了！

使用 data 对象、方法还是计算属性？

已经了解了 data 对象，可以使用它来存储字符串、数组和对象等数据；也了解了方法，可以使用它来存储函数并在模板中调用；还了解了计算属性，可以使用它将函数存储下来，然后像访问 data 对象中的属性一样调用。但是，应该使用哪一个？又应该在何时使用呢？

上述每一种选择都有适合的场景，而且最好和其他方式组合使用。但是对于一些特定任务而言，有的方式要比另外一些更好。例如，如果需要接收一个参数，那么肯定要用方法，而不是 data 对象或者计算属性：这两者都不能接收参数。

data 对象最适合纯粹的数据：如果想将数据放在某处，然后在模板、方法或者计算属性中使用，那么可以把它放在 data 对象中。后面也许还会更新它。

当你希望为模板添加函数功能时，最好使用方法：给方法传递数据，然后它们会对数据进行处理，最终可能返回不同的数据结果。

计算属性适用于执行更加复杂的表达式，这些表达式往往太长或者需要频繁地重复使用，所以不想在模板中直接使用。计算属性往往和其他计算属性或者 data 对象一起使用，基本上就像是 data 对象的一个扩展和增强版本。

可以在表格 1-1 中比较这些处理数据的方法—— data 对象、方法和计算属性。

表格 1-1：data 对象 vs 方法 vs 计算属性

	可读？	可写？	可以接受参数？	需要运算？	有缓存？
data 对象	是	是	否	否	无效，因为它不需要运算
方法	是	否	是	是	否
计算属性	是	是	否	是	是

这个表格可能看起来令人生畏，但它可以帮助你快速掌握使用的诀窍。

侦听器

侦听器可以监听 data 对象属性或者计算属性的变化。

如果从其他框架切换到 Vue ，那么你可能一直好奇如何监听数据的变化，而且一直期待着这个功能。但是，要小心。在 Vue 中，通常有比侦听器更好的方式来处理问题——通

常会使用计算属性。例如，和设置数据然后监听它的变化相比，使用一个带有 setter 的计算属性会是更好的方式。

侦听器使用起来很简单：只要设置要监听的属性名就可以。例如下面这个例子监听了 this.count 的变化：

```
<script>
  new Vue({
    el: '#app',
    data: {
      count: 0
    },
    watchers: {
      count() {
        // this.count 有变化了!
      }
    }
  });
</script>
```

尽管大部分简单的例子用不到侦听器，但侦听器很适合用于处理异步操作。假设我们有一个可以让用户输入的输入框，但是想将 5 秒前的输入内容显示到页面上。要做到这一点，可以使用 v-model 将输入框的值绑定到 data 对象中的一个属性，然后监听该属性的变化，并且在侦听器中将这个值在一定的延迟之后赋值给 data 对象的另一个属性：

```
<div id="app">
  <input type="text" v-model="inputValue">
  <p>5 秒之后，显示输入的值为 "{{ oldInputValue }}".</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      inputValue: '',
      oldInputValue: ''
    },
    watch: {
      inputValue() {
        const newValue = this.inputValue;
        setTimeout(() => {
```


```
            this.oldInputValue = newValue;
          }, 5000);
        }
      }
    });
  </script>
```

上面这个例子中需要注意一点，我们在侦听器函数中将 this.inputValue 赋值给了一个局部变量：因为不这样做，当传递给 setTimeout 的函数被调用时，this.inputValue 有可能已经被更新到最新值了！

监听 data 对象中某个对象的属性

有些时候会将一整个对象存储在 data 对象中。为了监听这个对象的属性变化，可以在侦听器的名称中使用 . 操作符，就像访问这个对象属性一样：

```
new Vue({
  data: {
    formData: {
      username: ''
    }
  },
  watch: {
    'formData.username'() {
      // this.formData.username 有变化了
    }
  }
});
```

获取旧值

当监听的属性发生变化时，侦听器会被传入两个参数：所监听属性的当前值和原来的旧值。这一特性可以用来了解到底发生了什么变化：

```
watch: {
  inputValue(val, oldVal) {
    console.log(val, oldVal);
  }
}
```

其中 val 等于 this.inputValue。我通常会直接使用后者。

27 深度监听

当监听一个对象时，可能想监听整个对象的变化，而不仅仅是某个属性。但在默认情况下，如果你正在监听 formData 对象并且修改了 formData.username，对应的侦听器并不会触发，它只在 formData 对象被整个替换时触发。

监听整个对象被称作深度监听，通过将 deep 选项设置为 true 来开启这一特性：

```
watch: {
  formData: {
    handler() {
      console.log(val, oldVal);
    },
    deep: true
  }
}
```

过滤器

过滤器是一种在模板中处理数据的便捷方式，而且经常会在其他模板语言中见到。它们特别适合对字符串和数字进行简单的显示变化：例如，将字符串变为正确的大小写格式，或者用更容易阅读的格式显示数字。

请看代码示例：

```
<div id="app">
  <p>商品一花费了：${{ (productOneCost / 100).toFixed(2) }}</p>
  <p>商品二花费了：${{ (productTwoCost / 100).toFixed(2) }}</p>
  <p>商品三花费了：${{ (productThreeCost / 100).toFixed(2) }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      productOneCost: 998,
      productTwoCost: 2399,
      productThreeCost: 5300
    }
  });
</script>
```


这段代码可以正常工作，但是存在很多重复。对于每一项商品我们都要进行下面的计算：将价格单位从分转换为元、格式化为两位小数，同时添加美元符号。尽管我们可以将这一逻辑拆分为一个方法，但这次我们要把这一逻辑拆分为过滤器，因为这样可读性更好而且可以全局使用：


```
<div id="app">
  <p>商品一花费了：{{ productOneCost | formatCost }}</p>
  <p>商品二花费了：{{ productTwoCost | formatCost }}</p>
  <p>商品三花费了：{{ productThreeCost | formatCost }}</p>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      productOneCost: 998,
      productTwoCost: 2399,
      productThreeCost: 5300
    },
    filters: {
      formatCost(value) {
        return '$' + (value / 100).toFixed(2);
      }
    }
  });
</script>
```

这样会好很多——代码重复更少、更加易读，同时可维护性也更好。如果现在决定对它增加新的逻辑——例如增加货币换算功能——只需要修改一次而不是修改每一处用到它的代码。

可以用链式调用的方式在一个表达式中使用多个过滤器。例如，我们有一个 round 过滤器可以将数字四舍五入到整数，那么可以用 {{ productOneCost | round | formatCost }} 的方式同时使用两个过滤器。首先会调用 round 过滤器，它返回的结果会传递给 formatCost 过滤器进行处理，然后输出到页面。

过滤器同样可以接收参数。在下面这个例子中，输入的字符串会作为第二个参数传递给过滤器函数：

```
<div id="app">
  <p>商品一花费了：{{ productOneCost | formatCost('$') }}</p>
```


```
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      productOneCost: 998,
    },
    filters: {
      formatCost(value, symbol) {
        return symbol + (value / 100).toFixed(2);
      }
    }
  });
</script>
```

29 除了在插值中使用，还可以在 v-bind 中使用过滤器（当绑定数值到属性时）：

```
<div id="app">
  <input type="text" v-bind:value="productCost | formatCost('$')">
</div>
```

也可以使用 Vue.filter() 来注册一个全局的过滤器，而不是将过滤器逐一注册到各个组件上：

```
Vue.filter('formatCost', function (value, symbol) {
  return symbol + (val / 100).toFixed(2);
});
```

这种方式适合注册整个应用中都会用到的过滤器。我一般把我所有的过滤器都放到单独的 filters.js 文件中。

使用过滤器有两个小的注意事项。一个是过滤器是组件中唯一不能使用 this 来访问数据或者方法的地方。这一点是故意设计成这样的：因为过滤器应该是纯函数，也就是说对于同样的输入每次都返回同样的输出，而不涉及任何外部数据。如果想在过滤器中访问其他数据，可以将它作为参数传入。

另外一个注意事项是只可以在插值和 v-bind 指令中使用过滤器。而在 Vue 1 中，可以在任何可以使用表达式的地方使用过滤器，例如在 v-for 指令中：

```
<li v-for="item in items | filterItems">{{ item }}</li>
```


Vue 2 取消了这种做法，对于上面这个例子必须使用方法或者计算属性。

使用 ref 直接访问元素

有时你会发现需要直接访问一个 DOM 元素；也许你正在使用一个不支持 Vue 的第三方库，或者希望做一些 Vue 自身不能完全处理的事情。可以使用 ref 直接访问元素，而不需要使用 querySelector 或者其他选择 DOM 节点的原生方法。

使用 ref 访问一个元素，只需要将这个元素的 ref 属性设置为字符串，然后可以使用这个字符串访问元素：

```
<canvas ref="myCanvas"></canvas>
```

这样，在 JavaScript 中，这个元素会被存储到 this.$ref 这个对象中，对应的键名就是为元素的 ref 属性设置的值。在这个例子中，可以使用 this.$ref.myCanvas 访问这个元素。

在组件中使用 ref 尤为有用。同一组件的代码可能在页面上出现多次，这意味着根本不能为组件内的元素添加一个唯一的类名然后使用 querySelector 来选择该元素。相比之下，this.$refs 只包含当前组件内部元素的引用，这意味着如果在组件内调用 this.$refs.blablabla，它总是指向该组件内的对应元素，而不是页面其他地方的元素。

输入和事件

到目前为止，你了解到的每个知识点几乎都只和数据显示有关——还没有涉及任何交互性的内容。下面介绍 Vue 中的事件绑定。

可以使用 v-on 指令将事件侦听器绑定到元素上。这个指令将事件名称作为参数，然后将事件侦听器作为传入值。例如，想要在按钮被单击时将 counter 的值增加 1，可以编写如下的代码：

```
<button v-on:click="counter++">单击增加计数</button>
<p>你已经单击了按钮 {{ counter }} 次</p> times.
```

也可以提供一个方法名，当按钮被单击时会调用该方法名对应的方法：

```
<div id="app">
  <button v-on:click="increase">单击增加计数</button>
```


```
    <p>你已经单击了按钮 {{ counter }} 次</p> times.
  </div>
  <script>
    new Vue({
      el: '#app',
      data: {
        counter: 0
      },
      methods: {
        increase(e) {
          this.counter++;
        }
      }
    });
  </script>
```

这个例子和前一个例子效果是一样的。

使用方法和内联代码的一个明显区别是，如果使用方法，事件对象会作为第一个参数传入。这个事件对象是原生的 DOM 事件对象，如果你使用 JavaScript 内建的 `.addEventListener()` 方法添加事件监听器，会得到相同的事件对象，而且它非常有用，例如可以获取键盘事件的 keyCode 值。

31 当使用内联代码时，也可以通过 `$event` 变量访问事件对象。当你将同一事件侦听器添加到多个元素，并希望了解是哪个元素触发的事件时，这一功能会很有用。

v-on 简写

和 `v-bind` 指令类似，`v-on` 指令同样有一个简写方式。可以将 `v-on:click` 简写为 `@click`。

下面这个例子和前一个例子是一样的：

```
<button @click="increase">单击增加计数</button>
```

我几乎总是用简写，比较少用 v-on。

事件修饰符

还可以使用很多修饰符来修改事件侦听器或者事件本身。

如果想要阻止执行事件默认行为——例如，当链接被单击时阻止页面的跳转——可以使用 .prevent 修饰符：

```
<a @click.prevent="handleClick">...</a>
```

如果想要阻止事件继续传播，以避免在父级元素上触发事件，可以使用 .stop 修饰符：

```
<button @click.stop="handleClick">...</button>
```

如果想要只在第一次触发事件的时候触发事件侦听器，可以使用 .once 修饰符：

```
<button @click.once="handleFirstClick">...</button>
```

如果想要使用捕获模式，也就是说，事件会在传递到当前元素的下级元素前触发（而在冒泡模式中，事件会先在当前元素上触发，然后沿 DOM 树向上冒泡），可以使用 .capture 修饰符：

```
<div @click.capture="handleCapturedClick">...</div>
```

如果想要只监听元素自身而不是它的子元素上触发的事件（也就是说，event.target 就是绑定该侦听器的元素时），可以使用 .self 修饰符：

```
<div @click.self="handleSelfClick">...</div>
```

也可以只设置事件名和修饰符而不传入侦听器，而且可以将修饰符串联起来使用。例如，下面这个例子会阻止单击事件沿 DOM 树向下传递，但只在第一次触发时有效：

```
<div @click.stop.capture.once></div>
```

32

除了上述事件修饰符，还有一些按键修饰符。它们用在键盘事件上，只有在特定按键按下时才会触发事件。看下面这个例子：

```
<div id="app">
  <form @keyup="handleKeyup">...</form>
</div>
<script>
  new Vue({
    el: '#app',
    methods: {
      handleKeyup(e) {
        if (e.keyCode === 27) {
          // 做点什么
```


```
        }
      }
    }
  });
</script>
```

在上面这个例子中，if 语句中的代码只有当 keyCode 为 27 的按键——也就是 Esc 键——被按下时才会执行。但是，Vue 内置了一种基于修饰符的方式来处理这种情况。可以将按键键值作为修饰符，例如下面这样：

```
<form @keyup.27="handleEscape">...</form>
```

现在，只有在 Esc 键被按下时才会触发 handleEscape 侦听器。Vue 还为最常用的按键提供了别名：.enter、.tab、.delete、.esc、.space、.up、down、.left 和 right。不想使用 @keyup.27 并且记住键值 27 所代表的含义的话，只要使用 @keyup.esc 就可以了。

Vue 从 2.5.0 版本开始，可以使用键盘事件对象中 key 属性的所有可能值作为修饰符。例如，按下左侧的 Shift 键时，e.key 等于 ShiftLeft。因此要想监听该 Shift 键何时松开，可以使用下面的代码：

```
<form @keyup.shift-left="handleShiftLeft">...</form>
```

和按键修饰符类似，有 3 个鼠标按钮修饰符可以添加到鼠标事件上：.left、.middle 和 .right。

还有一些用于 Ctrl 和 Shift 等系统修饰键的修饰符：.ctrl、.alt、.shift 和 .meta。前 3 个修饰符可以很简单地从名字了解其含义，但 .meta 却不是很明确：在 Windows 系统中，它代表 Windows 键；在 macOS 中，它代表 Command 键。

最后，如果想在只有被指定的按键被按下但没有其他按键被按下的时候才触发事件侦听器，可以使用 .exact 修饰符。例如：

```
<input @keydown.enter.exact="handleEnter">
```

33 在上面这个例子中，当 Enter 键被按下，且没其他任何按键——例如 Command-Enter 或者 Ctrl-Enter——被按下时，侦听器才会被触发。

生命周期钩子

到现在为止书中已经有几处出现这种用法了：如果将一个函数设置为一个组件或者 Vue 实例的 created 属性，它会在组件创建完成时被调用。这是生命周期钩子的一个例子。生命周期钩子是一系列会在组件生命周期——从组件被创建并添加到 DOM，到组件被销毁的整个过程——的各个阶段被调用的函数。

虽然 Vue 有 8 个生命周期钩子，但是很容易记住，因为其中 4 个是带有 before 前缀的钩子，它们会先于其他钩子被调用。

来看一下基本的生命周期：首先，Vue 实例会在执行 new Vue() 时初始化。此时，第一个钩子函数 beforeCreate 会被调用，同时响应式功能会被初始化，然后 created 钩子函数会被调用——从钩子的名字就能明白它们的执行机制：带有“before”前缀的钩子会在相关工作开始之前被调用，然后，真正执行相关工作的钩子才会被触发。接下来会解析模板——模板内容可以从 template 或 render 选项获取，或者从 Vue 初始化时所挂载元素的 outerHTML 获取。此时，就可以开始创建 DOM 元素了，因此会先触发 beforeMount 钩子，然后创建 DOM 节点，之后再触发 mounted 钩子。

有一点需要注意，在 Vue 2.0 中，mounted 钩子触发时并不保证元素已经被添加到 DOM 上。如果想保证元素已经被添加，可以调用 Vue.nextTick() 方法（也可以通过 this.$nextTick() 调用）并传入一个回调函数，在回调函数中添加需要在元素被添加到 DOM 之后运行的代码。例如：

```
<div id="app">
  <p>Hello world</p>
</div>
<script>
  new Vue({
    el: '#app',
    mounted() {
      // 元素可能还没有添加到 DOM 上

      this.$nextTick(() => {
        // 确定元素已经被添加到 DOM 上了
      });
    }
  });
</script>
```


到此为止，已经有 4 个钩子函数被触发，此时组件实例已经被初始化并添加到 DOM 上，并且用户已经可以看到我们的组件。也许我们的数据会更新，但是 DOM 也会被更新以响应数据的变化。在处理更新前，beforeUpdate 钩子函数会被触发，而且在应用更新之后，updated 钩子会被触发。当数据多次变化时，这一钩子函数可以被多次触发。

我们已经看到了组件的创建和运行，现在终于到了组件“离开”的时候了。在组件从 DOM 上被移除前，beforeDestroy 钩子会被触发，并且在它被移除后，destroyed 钩子会被触发。

这就是在 Vue 实例的生命周期中会触发的所有钩子函数。现在，我们从这些钩子本身而不是 Vue 实例的角度再将它们梳理一遍：

- beforeCreate 在实例初始化前被触发。
- created 会在实例初始化之后、被添加到 DOM 之前触发。
- beforeMount 会在元素已经准备好被添加到 DOM，但还没有添加的时候触发。
- mounted 会在元素创建后触发（但并不一定已经添加到了 DOM，可以用 nextTick 来保证这一点）。
- beforeUpdate 会在由于数据更新将要对 DOM 做一些更改时触发。
- updated 会在 DOM 的更改已经完成后触发。
- beforeDestroy 会在组件即将被销毁并且从 DOM 上移除时触发。
- destroyed 会在组件被销毁后触发。

尽管看起来确实有很多钩子，但你只需要记住 4 个（created、mounted、updated 和 destroyed），然后可以推导出其他 4 个。

自定义指令

除了 v-if、v-model 以及 v-html 等内置的指令，还可以创建自定义指令。当你想直接对 DOM 进行某些操作时，指令非常好用——如果发现自己并不需要访问 DOM，那么使用不带指令的组件就好。

举一个简单的例子，来实现一个 v-blink 指令，用来模拟 <blink> 标签的行为。可以像下面这样来使用它：

```
<p v-blink> 本段内容会闪烁 </p>
```


添加一个指令类似于添加一个过滤器：可以在将它传入 Vue 实例或者组件的 directives 属性，或者使用 Vue.directive() 注册一个全局指令。需要传入指令的名字，以及一个包含钩子函数的对象，这些钩子函数会在设置了该指令的元素的生命周期的各个阶段运行。

一共有 5 个钩子函数，分别是 bind、inserted、update、componentUpdated 和 unbind。一会儿详细介绍它们，但目前只用 bind 钩子，这个钩子会在指令绑定到元素上时被调用。在这个钩子中，我们会每隔一秒切换一次元素的可见性：

```
Vue.directive('blink', {
  bind(el) {
    let isVisible = true;
    setInterval(() => {
      isVisible = !isVisible;
      el.style.visibility = isVisible ? 'visible' : 'hidden';
    }, 1000);
  }
});
```

现在，任何使用了该指令的元素都会每秒闪烁一次——这正是我们想要的。

指令有多个钩子函数，正如 Vue 实例和组件有生命周期钩子一样。它们有不同的命名，和生命周期钩子并不完全一样，所以现在仔细分析下：

- bind 钩子函数会在指令绑定到元素时被调用。
- inserted 钩子会在绑定的元素被添加到父节点时被调用——但和 mounted 一样，此时还不能保证元素已经被添加到 DOM 上。可以使用 this.$nextTick 来保证这一点。
- update 钩子会在绑定该指令的组件节点被更新时调用，但是该组件的子组件可能此时还未更新。
- componentUpdated 钩子和 updated 钩子类似，但它会在组件的子组件都更新完成后调用。
- unbind 钩子用于指令的拆除，当指令从元素上解绑时会被调用。

不必每次都调用所有的钩子。事实上，它们都是可选的，所以它们中的任何一个都不是必须要调用的。

我自己最常使用 bind 和 update 钩子。方便起见，如果只想用这两个钩子，可以采用一

种快捷方法——可以不用传入一个对象，而是传入一个函数作为参数，这个函数会分别被这两个钩子调用：

```
Vue.directive('my-directive', (el) => {
  // 这段代码会分别被 "bind" 和 "update" 这两个钩子所调用
});
```

36 钩子函数参数

在上文中你已经见过指令可以接收参数（v-bind:class）、修饰符（v-on.once）和值（v-if="expression"）。可以通过传入钩子函数的第二个参数——binding——来访问所有的这些内容。

如果通过 v-my-directive:example.one.two="someExpression" 来使用指令，那么 binding 对象会包含以下属性：

- name 属性是指令的名称，但不包含 v-。在这个例子中，name 的值为 my-directive。
- value 属性是传入指令的值。在这个例子中，它将会是 someExpression 表达式的计算值。例如，假设 data 对象等于 { someExpression: hello world }，那么 value 的值就是 hello world。
- oldValue 属性是上一次传入指令的值，它只有在 update 和 componentUpdated 钩子函数中才可以使用。在例子中，如果改变 someExpression 的值，update 钩子就会被调用，此时 value 属性为新的值，oldValue 属性为旧的值。
- expression 属性是指令表达式的字符串形式，之后会对该表达式求值。在这个例子中，它的值为 someExpression。
- arg 属性是传入指令的参数，在这个例子中也就是 example。
- modifiers 属性是一个包含所有传入指令的修饰符的对象。在这个例子中它等于 { one: true, two: true }。

为了更好地解释如何使用参数，来修改前面示例中的指令，让它接收一个参数，用于表示元素闪烁的频率：

```
Vue.directive('blink', {
  bind(el, binding) {
    let isVisible = true; setInterval(() => {
      isVisible = !isVisible;
```


```
        el.style.visibility = isVisible ? 'visible' : 'hidden';
      }, binding.value || 1000);
    }
  });
```

上面的代码中添加了 binding 参数，并且将 setInterval 的计时时间设置为 binding.value || 1000 而不只是 1000。但是这个例子缺少了用于处理该指令值变化后更新组件的逻辑。为了实现这一点，需要将 setInterval 返回的 ID 存储到元素的一个 data 属性上，然后在 update 钩子中取消旧的计时器，之后再创建一个新的定时器。

过渡和动画

Vue 提供了大量功能来为你的 Vue 应用增加动画和过渡效果——从在元素进入、离开页面或者修改时使用 CSS 过渡和 JavaScript 动画，到实现多个组件间的过渡，甚至数据元素本身的动效。对于一本讲解 Vue 的入门书而言，这会涉及过多的知识点，因此我会集中讲解我自己最常使用的内容，你可以稍后仔细阅读 Vue 关于过渡部分的文档（里面有一些很棒的示例）。

首先，研究下 <transition> 组件，以及如何使用它和 CSS 过渡来给元素添加进入 / 离开页面的动画。

CSS 过渡

CSS 过渡对于简单的动画效果很好用，在 CSS 过渡中你只是将一个或多个 CSS 属性从一个值过渡到另一个值。例如，可以将 color 从 blur 过渡到 red，或者将 opacity 从 1 过渡到 0。Vue 提供了 <transition> 组件，它会向内部的带有 v-if 指令的元素添加类名，因此可以使用这个组件来为进入或离开的元素添加 CSS 过渡动画。

在这一章中，我们都会使用下面的模板作为例子，这个模板中有一个按钮，和一个会在按钮被单击时切换显示状态的元素：

```
<div id="app">
  <button @click="divVisible = !divVisible">切换可见性</button>

  <div v-if="divVisible">本段内容时而隐藏时而可见</div>
</div>
<script>
  new Vue({
```


```
    el: '#app',
    data: {
      divVisible: true }
  });
</script>
```

现在如果单击按钮，div 元素的内容会立刻隐藏或者再次显示，但没有过渡效果。

假设想要给它添加一个简单的过渡：在切换显示状态时让 div 元素渐隐或渐现。要实现这一功能，可以像下面这样将 div 包裹在 transition 组件中：

```
<transition name="fade">
  <div v-if="divVisible">This content is sometimes hidden</div>
</transition>
```

38 仅用上面的代码并不会有任何效果（它的表现和添加 <transition> 元素前是完全一样的），但添加下面的 CSS，就可以得到渐变过渡效果：

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to {
  opacity: 0;
}
```

现在，当单击按钮切换显示状态时，元素会在页面上渐现或渐隐，而不像之前那样立即被添加或移除。

它的工作原理是，Vue 获取 tansition 组件的 name 属性值，然后使用它在过渡的各个节点为包含的元素添加类名。当元素被添加到文档或者从文档中移除时，会分别应用 enter 和 leave 两类过渡。以下是会添加的类名：

`{name}-enter`

这个类名会在元素被插入 DOM 时加入，然后在下一帧立刻移除。可以使用它来设置那些需要在元素开始进入过渡时移除的 CSS 属性。

`{name}-enter-active`

这个类名会在元素整个动画阶段应用。它和 `-enter` 类名同时被添加，然后在动画完成时被移除。这个类适用于设置 `transition` 这个 CSS 属性，以设置过渡的时间

长度、过渡的属性和使用的曲线函数。

{name}-enter-to

这个类名会在 -enter 类名从元素上移除的同时添加到元素上。它适合用来设置那些在元素开始进入过渡时添加的 CSS 属性，但我通常发现，在 -enter 类上设置与之相反的过渡属性会更好用一些。

{name}-leave

在离开过渡中，这个类名相当于进入过渡中的 -enter 类名。它在离开过渡触发时被添加，然后在下一帧被移除。和 -enter-to 类名类似，这个类名不太有用：更好的做法是使用 -leave-to 进行与之相反的过渡动画。

{name}-leave-active

在离开过渡中，这个类名相当于进入过渡中的 -enter-active。它应用于离开过渡的整个阶段。

{name}-leave-to

在离开过渡中，这个类名相当于进入过渡中的 -enter-to。它在离开过渡被触发之后下一帧生效（与此同时 -leave 被删除），在过渡完成之后才被移除。

在实践中，我发现自己最常使用下面 4 个类名：

{name}-enter

使用这个类名设置在进入过渡阶段需要过渡的 CSS 属性。

{name}-enter-active

使用这个类名设置进入过渡的 transition CSS 属性。

{name}-leave-active

使用这个类名设置离开过渡的 transition CSS 属性。

{name}-leave-to

使用这个类名设置在离开过渡阶段需要过渡的 CSS 属性。

因此，前面的示例代码会实现这样的效果：进入过渡和离开过渡的时间长度都是 0.5s，需要过渡的属性为 opacity（使用默认的过渡曲线），而且在进入的时候，opacity 的值从 0 过渡到 1，而离开的时候，则从 1 过渡到 0。因此当元素进入文档时会有淡入的效果，而离开时又会有淡出的效果。

JavaScript 动画

除了 CSS 动画，<transition> 组件还提供了用于实现 JavaScript 动画的钩子。使用这些钩子，可以使用自己的代码或者类似于 GreenSock 或者 Velocity 的库来实现动画。

这些钩子和用于 CSS 过渡的类名类似：

beforeEnter

这个钩子会在进入动画开始前被触发，适合设置初始值。

enter

这个钩子会在进入动画开始时被触发，可以在这里运行动画。可以使用 done 回调来标明动画已经完成。

afterEnter

这个钩子会在进入动画执行完成时被触发。

enterCancelled

这个钩子会在进入动画被取消时触发。

40 beforeLeave

这个钩子对于离开动画而言等同于进入动画的 beforeEnter 钩子，会在离开动画开始前被调用。

leave

这个钩子对于离开动画而言等同于进入动画的 enter 钩子，可以在这里运行进入动画。

afterLeave

这个钩子会在离开动画执行完成时被触发。

leaveCancelled

这个钩子会在离开动画被取消时触发。

这些钩子会以事件的形式在 transition 组件上触发，像下面这样：

```
<transition
  v-on:before-enter="handleBeforeEnter"
  v-on:enter="handleEnter"
  v-on:leave="handleLeave">
  <div v-if="divVisible">...</div>
</transition>
```

通过添加这些事件处理函数，可以使用 GreenSock 添加和 CSS 过渡示例相同的效果，像下面这样：

```
new Vue({
  el: '#app',
  data: {
    divVisible: false
  },
  methods: {
    handleBeforeEnter(el) {
      el.style.opacity = 0;
    },
    handleEnter(el, done) {
      TweenLite.to(el, 0.6, { opacity: 1, onComplete: done });
    },
    handleLeave(el, done) {
      TweenLite.to(el, 0.6, { opacity: 0, onComplete: done });
    }
  }
});
```

使用 JavaScript 动画，可以创建比使用 CSS 过渡复杂得多的动画效果，包括多步动画、或者每次显示不同的过渡效果。但是 CSS 过渡通常性能更好，所以尽量使用 CSS 过渡，除非你需要的效果无法用纯 CSS 过渡实现。

到此为止，我们已经探讨了 Vue 的一些基础用法，接下来会介绍如何使用组件来组织代码。

41 总结

在这一章中，我们着眼于 Vue 的一些基础用法：

- 了解了一些使用 Vue 的原因。
- 了解了如何使用 CDN 或者 webpack 安装和配置 Vue。
- 了解了 Vue 的语法：如何使用模板、data 对象和指令将数据显示到页面上。
- 了解了 `v-if` 指令和 `v-show` 指令的区别。
- 了解了如何使用 `v-for` 在模板中进行循环。
- 了解了如何使用 `v-bind` 指令将 data 对象的属性绑定到 HTML 元素的属性上。
- 了解了 Vue 如何在数据更新时自动地更新页面上显示的内容：这一特性被称作响应式。
- 了解了双向数据绑定：使用 `v-model` 将数据显示在输入框中，以及当输入框的输入值被改变时更新 data 对象。
- 了解了如何使用 `v-html` 指令将 data 对象中的数据直接设置为一个元素内部的 HTML 内容。
- 了解了如何使用方法，从而可以在模板中和整个 Vue 实例中访问函数。同时我们还了解了方法中 `this` 的指向。
- 了解了如何使用计算属性来创建可访问的值，就像访问 data 对象的属性一样，但是它们是在运行时计算得到的，并且是以函数的形式被定义的。
- 了解了如何使用侦听器来监听 data 对象的属性或计算属性的变化，并且在变化发生时进行一些处理——但是通常情况下应该避免使用侦听器，计算属性往往是不错的选择。
- 了解了过滤器，一种在模板中处理数据的便捷方法——例如，格式化数据。
- 了解了如何使用 `ref` 直接访问元素，在使用 Vue 不支持的第三方库或者是做一些 Vue 自身不支持的操作时，可以使用它。
- 了解了使用 `v-on` 实现事件绑定，以及它的简写语法：在事件名称前添加 `@` 前缀。
- 了解了 Vue 实例的生命周期，以及如何在钩子函数中运行代码。
- 了解了如何创建自定义指令。
- 了解了如何使用 Vue 提供的 CSS 过渡和 JavaScript 动画功能。

第 2 章

Vue.js 组件

到现在为止，你已经看到书中多次提到了组件。那么什么是组件呢？组件是一段独立的、代表了页面的一个部分的代码片段。它拥有自己的数据、JavaScript 脚本，以及样式标签。组件可以包含其他的组件，并且它们之间可以相互通信。组件可以是按钮或者图标这样很小的元素，也可以是一个更大的元素，比如在整个网站或者整个页面上重复使用的表单。

将代码从页面中分离到组件中的主要优势是，负责页面每一部分的代码都很靠近该组件中的其余代码。因此当你想要知道哪个元素有添加事件监听器，不必再在一堆 JavaScript 文件中搜索相应的选择器，因为 JavaScript 代码就在对应的 HTML 旁边！而且由于组件是独立的，还可以确保组件中的代码不会影响任何其他组件或产生任何副作用。

组件基础

直接来演示一个简单的组件：

```
const CustomButton = {
  template: '<button>自定义按钮</button>'
};
```

这样就完成了。可以通过 components 配置对象，将这个组件传入你的 app：

```
<div id="app">
  <custom-button></custom-button>
</div>
<script>
```


```
    const CustomButton = {
      template: '<button>自定义按钮</button>'
    };

    new Vue({
      el: '#app',
      components: {
        CustonButton
      }
    });
  </script>
```

这样自定义的按钮就会在页面上显示了。

也可以注册一个全局的组件，只需像下面这样调用 Vue.component() 方法：

```
Vue.component('custom-button',{
  template: '<button>自定义按钮</button>'
});
```

然后就可以在模板中使用这个组件了，和前面示例中的使用方式相同，但是你不需要在 components 配置对象中指定它了，它现在随处可用。

数据、方法和计算属性

每个组件可以拥有它们自己的数据、方法和计算属性，以及所有在前面章节中出现过的属性——就像 Vue 实例一样。定义组件的对象与我们用来定义 Vue 实例的对象相似，它们在很多地方可以互用。比如说，我们可以定义并注册一个包含数据和计算属性的全局组件：

```
Vue.component('positive-numbers', {
  template: '<p>有{{ positiveNumbers.length }}个正数</p>',
  data() {
    return {
      numbers: [-5, 0, 2, -1, 1, 0.5]
    };
  },
  computed: {
    positiveNumbers() {
      return this.numbers.filter((number) => number >= 0);
    }
```

```
  }
});
```

然后就可以在 Vue 模板的任何地方，通过 <positive-numbers></positive-numbers> 标签来使用这个组件。

或许你已经注意到了组件与 Vue 实例之间的一个细微差别：Vue 实例中的 data 属性是一个对象，然而组件中的 data 属性是一个函数。这是因为一个组件可以在同一个页面上被多次引用，你大概不希望它们共享一个 data 对象——想象一下，单击了一个按钮组件，同时在页面另一侧的按钮组件也做出了响应（译者注：因为同一个组件的每个实例的 data 属性是同一个对象的引用，当该组件的某个实例修改了自身的 data 属性，相当于所有实例的 data 属性都被修改了）！所以组件的 data 属性应该是一个函数，在组件初始化时 Vue 会调用这个函数来生成 data 对象。如果忘记将组件的 data 属性设置为函数，Vue 会抛出一个警告。

传递数据

组件很有用，但当你开始传递数据到它内部时，组件才真正地展示出力量。可以使用 *props* 属性来传递数据，示例如下：

```
<div id="app">
  <color-preview color="red"></color-preview>
  <color-preview color="blue"></color-preview>
</div>
<script>
  Vue.compoent('color-preview', {
    template: '<div class="color-preview" :style="style"></div>',
    props: ['color'],
    computed: {
      style() {
        return { backgroundColor: this.color };
      }
    }
  });

  new Vue({
    el: '#app'
  });
</script>
```


Props 是通过 HTML 属性传入组件的（比如示例中的 `color="red"`）；之后在组件内部，*props* 属性的值表示可以传入组件的属性的名称——这个例子中，就只有 `color`。然后，在组件内部就可以通过 `this.color` 来获取属性的值了。以上示例代码会输出如下 HTML 片段：

```
<div id="app">
  <div class="color-preview" style="background-color: red"></div>
  <div class="color-preview" style="background-color: blue"></div>
</div>
```

Prop 验证

除了可以传递一个的简单数组，来表明组件可以接收的属性的名称，也可以传递一个对象，来描述属性的信息，比如它的类型、是否必须、默认值以及用于高级验证的自定义验证函数。

要指定一个 prop 的类型，可以为它传递一个原生的构造函数，例如 `Number`、`String` 或者 `Object`，也可以是一个用于 instanceof 操作符检测的自定义构造函数。例如：

```
Vue.component('price-display',{
  props: {
    price: Number,
    unit: String
  }
});
```

如果 `price` 不是一个数字，或者 `unit` 不是一个字符串，Vue 就会抛出一个警告。

如果一个 prop 可以是多个类型中的一个，你就可以为它传递一个包含所有有效类型的数组，例如 `price: [Number, String, Price]`（这里的 `Price` 是一个自定义的构造函数）。

也可以指定一个 prop 是否是必需的，或者在没有传入值时，给它设定一个默认值。为此，可以传递给它一个对象而不是像前面那样的构造函数，并通过该对象的 `type` 属性来设置 prop 的类型：

```
Vue.component('price-display', {
  props: {
    price: {
      type: Number,
      required: true
```


```
    },
    unit: {
      type: String,
      default: '$'
    }
  }
})
```

在这个示例中，price 是一个必需的 prop，如果没有传递值给它，就会抛出警告。unit 不是必需的，但是有个默认值 $，如果你没有传入任何值，在 component 内 this.unit 将会等于 $。

最后，你可以传递一个验证函数，该函数以 prop 的值为参数，在 prop 有效时应该返回 true，而无效时则返回 false。例如下面的例子，验证了 price 是否大于零，这样你就不会在无意间为商品设置一个负数的价格：

```
price: {
  type: Number,
  required: true,
  validator(value) {
    return value >= 0;
  }
}
```


Prop 的大小写

Vue 处理 prop 的大小写的方式很友好：也许你只想在 HTML 中使用 kebab 形式（my-prop=""），而不想在 JavaScript 中使用 this['my-prop'] 来引用 prop 的值。这个时候 camel 形式（this.*myProp*）会更加方便书写和阅读。

幸运的是，Vue 都为我们处理好了。在 HTML 中通过 kebab 形式指定的属性，会在组件内部自动转换为 camel 形式：

```
<div id="app">
  <price-display percentage-discount="20%"></price-display>
</div>
<script>
  Vue.component('price-display',{
    props: {
      percentageDiscount: Number
```


```
    }
  });

  new Vue({
    el: '#app'
  });
</script>
```

我们不需要做任何额外的处理，这种转换就会自动完成。

响应式

你应该已经发现了，对于 data 对象、方法还有计算属性，当它们的值发生变化时，模板也会更新；同样，props 也是这样的。在父级实例中设定 prop 的值时，可以使用 v-bind 指令将该 prop 与某个值绑定。那么无论何时只要这个值发生变化，在组件内任何使用该 prop 的地方都会更新。

举个简单例子，来创建一个组件，它会显示设置在某个属性上的数字：

```
Vue.component('display-number',{
  template: '<p>当前数字是 {{ number }}</p>',
  props: {
    number: {
      type: Number,
      required: true
    }
  }
});
```

48 然后将这个组件显示到页面，同时将它的值每秒加 1：

```
<div id="app">
  <display-number v-bind:number="number"></display-number>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      number: 0
    },
    created(){
```


```
      setInterval(() => {
        this.number++;
      }, 1000);
    }
  });
</script>
```

传递给 display-number 的数字会每秒增加 1（记得吗？ created 函数在实例创建后执行），于是对应的 prop 也会每秒变化一次。Vue 很聪明，知道这些变化，所以它会更新页面。

如果 prop 的值不是字符串，那么就必须使用 v-bind 指令。比如下面的示例代码会抛出一个警告：

```
<display-number number="10"></display-number>
```

这是因为传递给 number 的是一个字符串，而不是数字。如果要传入一个数字，就要使用 v-bind 指令，它会把传入的值当作表达式求值，然后再传递给 prop：

```
<display-number v-bind:number="10"></display-number>
```

数据流和 .sync 修饰符

数据通过 prop 从父级组件传递到子组件中，当父级组件中的数据更新时，传递给子组件的 prop 也会更新。但是你不可以在子组件中修改 prop。这就是所谓的单向下行绑定，防止子组件在无意中改变父级组件的状态。

然而，双向数据绑定在某些情况下可能很有用。如果想要使用双向绑定，可以使用一个修饰符来实现：.sync 修饰符。它只是一个语法糖，请看示例：

```
<count-from-number
  :number.sync="numberToDisplay"
  />
```

上面的代码等效于：

49

```
<count-from-number
  :number="numberToDisplay"
  @update:number="val => numberToDisplay = val"
  />
```

所以，如果想要更改父级组件的值，需要触发 update:number 事件，该指令的参数——示例中为 number——是将要更新的值的名称。

让我们一起思考，如何实现一个 CountFromNumber 组件，它获取一个初始值并开始计数，同时更新父级组件的值：

```
Vue.component('count-from-number',{
  template: '<p>当前数字是 {{ number }}</p>',
  props: {
    number: {
      type: Number,
      required: true
    }
  },
  mounted(){
    setInterval(() => {
      this.$emit('update:number', this.number + 1);
    }, 1000);
  }
});
```

组件内部的值根本就不会变化，但是它改变了父级组件的值，这个值会通过 prop 传递回子组件。

在某些情况下，将触发事件的逻辑封装到计算属性中会有利于代码的组织，如下所示：

```
Vue.component('count-from-number', {
  template: '<p>当前数字是 {{ localNumber }}</p>',
  props: {
    number: {
      type: Number,
      required: true
    }
  },
  computed: {
    localNumber: {
      get() {
        return this.number;
      },
      set(value) {
        this.$emit('update:number', value);
      }
    }
  },
```


```
  mounted() {
    setInterval(() => {
      this.localNumber++;
    }, 1000);
  }
});
```

这样，localNumber 在效果上等同于 number。它获取 prop 的值，并且为它设置新的值时，会触发事件来更新父级组件的值（父级组件中更新后的值会再次传递到子组件中）。

需要注意的是，这样有可能引发无限循环：如果父级组件与子组件都对同一个值的更新做出反应，并且在处理更新的过程中再次改变这个值，这会让你的应用程序出现问题！

如果仅仅想要更新从 prop 传入的值，而不关心父级组件的值的更新，你可以在一开始的 data 函数中通过 this 来引用 prop 的值，将它复制到 data 对象中，就像这样：

```
Vue.component('count-from-number', {
  template: '<p>当前数字是 {{ number }}</p>',
  props: {
    initialNumber: {
      type: Number,
      required: true
    }
  },
  data() {
    return {
      number: this.initialNumber
    };
  },
  mounted() {
    setInterval(() => {
      this.number++;
    }, 1000);
  }
});
```

需要注意的是，如果 prop 的值更新了，组件内部并不会更新，因为它引用的是另外一个

值。如果你想要根据新提供的数值重新开始计数，可以为 initialNumber 添加一个侦听器，将新的值复制给 number。

51 自定义输入组件与 v-model

与 .sync 修饰符相似，可以在组件上使用 v-model 指令来创建自定义输入组件。这里同样也是一个语法糖。请看示例：

```
<input-username
  v-model="username"
  />
```

上面的代码等效于：

```
<input-username
  :value="username"
  @input="value => username=value"
  />
```

所以，为了创建 InputUsername 组件，我们需要它做两件事情：首先，它需要通过 value 属性获取初始值，然后不论何时只要 value 的值发生变化，它必须触发一个 input 事件。在这个例子中，我们要让组件将 value 的值转化为小写形式，再通过事件将它传递出去。

在之前例子中使用的方法（通过触发事件来改变值或使用计算属性）将不再有效。这里，必须监听输入框元素的 input 事件：

```
Vue.component('input-username', {
  template: '<input type="text" :value="value" @input="handleInput">',
  props: {
    value: {
      type: String,
      required: true
    }
  },
  methods: {
    handleInput(e) {
      const value = e.target.value.toLowerCase();

      // 如果 value 发生变化，input 的值也要更新
```


```
        if(value !== e.target.value) {
          e.target.value = value;
        }

        this.$emit('input',value);
      }
    }
});
```

现在可以像使用 input 元素一样使用这个组件了，v-model 的用法还是不变。唯一的区别是无法输入大写字母！

上面的示例存在一个问题：如果输入一个大写字母，光标会移动到文字的末尾。这在第一次输入内容时并不会有什么问题。但是如果回退光标去修改内容，光标将会出现跳动。

这是一个很容易解决的问题，但是它不在本示例的范围内；只需在设置 e.target.value 的值之前，保存当前光标的位置，在做出修改之后，再次设置光标的位置。

52

使用插槽（slot）将内容传递给组件

除了将数据作为 prop 传入到组件中，Vue 也允许传入 HTML。假如想要创建一个自定义按钮元素，你可能会通过一个属性来设置按钮的文本：

```
<custom-button text="单击我！"></custom-button>
```

这样可以满足需求，不过下面这种方式会更加自然：

```
<custom-button>单击我！</custom-button>
```

要在组件中使用这个文本内容，可以像下面这样使用 <slot> 标签：

```
Vue.component('custom-button', {
  template: '<button class="custom-button"><slot></slot></button>'
})
```

这样会生成如下 HTML 片段：

```
<button class="custom-button">单击我！</button>
```

不仅可以传入字符串，也可以传入任何你想要的 HTML，甚至是其他的 Vue 组件。这样可以创建复杂的页面，而不至于让组件的体积变得过于庞大。例如，可以把页面分割为一个头部组件、一个侧边栏组件和一个内容组件，模板可以像下面这样：

```
<div class="app">
  <site-header></site-header>

  <div class="container">
    <site-sidebar>
      ... 这里是侧边栏的内容 ...
    </site-sidebar>

    <site-main>
      ... 这里是正文的内容 ...
    </site-main>
  </div>
</div>
```

53 这是一个构建网站的好方法，特别是当你开始使用局部 CSS 和 vue-router 时，这两者会在本书的后面内容中介绍。

默认内容

如果为 `<slot>` 元素设定了内容，那么该内容会在组件没有接收到内容时被当作默认内容使用。让我们为之前看到的 `<custom-button>` 组件设置一些文本作为默认内容：

```
Vue.component('custom-button', {
  template: `<button class="custom-button">
    <slot><span class="default-text">默认的按钮文本</span></slot>
  </button>`
});
```

在 template 的字符串中编写 HTML 会显得笨重，所以再来看看这个例子，不过这次是在 HTML 代码中：

```
<button class="custom-button">
  <slot>
    <span class="default-text">默认的按钮文本</span>
  </slot>
</button>
```


可以看到，<slot> 元素内有默认的文本，而且文本还被一个有着 default-text 类名的 span 元素包裹着。这完全是可选的——没有必要将插槽元素中的内容或是默认内容用其他元素包裹起来；你可以使用任何你想要的 HTML 内容。

在“vue-loader 与 .vue 文件”一节中，将会讲到如何将 HTML 代码从组件对象中分离出去。template 属性只在非常简单的例子中才显得有用。

具名插槽

在前面的部分中看到的是*单个插槽*。这可能是插槽最普遍的用法，当然也是最容易理解的：传递给组件的内容会替换掉它里面的 <slot> 元素输出到页面上。

除此之外，还有具名插槽。*具名插槽* 具有——你已经猜到了——一个名称，它允许你在同一个组件中拥有多个插槽。

假设，有一个简单的博文组件，它有一个头部（通常是一个标题，但也可能是其他东西）和一些文本。这个组件的模板可以是下面这样的：

```
<section class="blog-post">
  <header>
    <slot name="header"></slot>
    <p>作者 {{ author.name }}</p>
  </header>

  <main>
    <slot></slot>
  </main>
</section>
```


然后在模板中引用这个组件时，可以使用 slot 属性来指定某个元素应该被插入名为 header 的插槽，而其他的 HTML 将被插入未命名的插槽：

```
<blog-post :author="author">
  <h2 slot="header">博文的标题</h2>

  <p>博文的内容</p>

  <p>更多内容</p>
</blog-post>
```


生成的 HTML 看起来会是下面这样的：

```
<section>
  <header>
    <h2> 博文的标题 </h2>
    <p> 作者 Callum Macrae</p>
  </header>

  <main>
    <p> 博文的内容 </p>

    <p> 更多内容 </p>
  </main>
</section>
```

作用域插槽

可以将数据传回 slot 组件，使父组件中的元素可以访问子组件中的数据。

创建一个获取用户信息的组件，而数据的显示则留给父级元素来处理：

```
Vue.component('user-data', {
  template: '<div class="user"><slot :user="user"></slot></div>',
  data: () => ({
    user: undefined,
  }),
  mounted() {
    // 设置 this.user...
  }
});
```

55 任何传递给 <slot> 的属性都可以用 slot-scope 属性中定义的变量来获取。引用这个组件，并用自己编写 HMTL 来显示用户信息：

```
<div>
  <user-data slot-scope="user">
    <p v-if="user"> 用户名：{{ user.name }}</p>
  </user-data>
</div>
```

这个功能与具名插槽组合在一起，用来覆盖元素的样式会很有用。来看一个显示文章摘要列表的组件：

```
<div>
  <div v-for="post in posts">
    <h1>{{ post.title }}</h1>
    <p>{{ post.summary }}</p>
  </div>
</div>
```

非常简单，它接收一个文章数组变量 `posts`，然后输出所有的文章标题和摘要。可以这样使用它：

```
<blog-listing :posts="posts"></blog-listing>
```

创建该组件的另一个版本。在这个版本中，可以传入自己的 HTML，用来显示文章的摘要——或许，举个例子，也有可能我们只想在页面上显示图片。那么在具名插槽元素中，就需要用一个段落元素将摘要信息包裹起来，之后只要愿意，就可以覆盖掉它。

新的组件看起来像这样：

```
<div>
  <div v-for="post in posts">
    <h1>{{ post.title }}</h1>

    <slot name="summary" :post="post">
      <p>{{ post.summary }}</p>
    </slot>
  </div>
</div>
```

现在，我们原来使用该组件的方式仍然有效，因为即使没有提供插槽元素，段落元素作为默认的内容仍然可以使用。但是如果愿意，可以覆盖掉摘要信息。

覆盖掉摘要信息来显示图片内容：

```
<blog-listing :posts="posts">
  <img
    slot="summary"
    slot-scope="post"
    :src="post.image"
    :alt="post.summary">
</blog-listing>
```

56

现在文本元素就被图片元素替换了，不过我们将文章的摘要作为图片的备用文字。这点很重要，它让使用屏幕阅读器等辅助技术的用户仍然可以读到文章的内容。

插槽作用域解构

作为一种简写方式，你可以解构 `slot-scope` 的属性，就像解构函数参数一样。用解构重写前面的例子：

```
<blog-listing>
  <img
    slot="summary"
    slot-scope="{ image, summary }"
    :src="image"
    :alt="summary">
</blog-listing>
```

自定义事件

除了可以处理原生 DOM 事件，`v-on` 指令也可以处理组件内部触发的自定义事件。调用 `this.$emit()` 函数可以触发一个自定义事件，它接收一个事件名称以及其他任何你想要传递的参数。然后就可以使用组件上的 `v-on` 指令来监听这个事件了。

下面是一个简单的组件，每次被单击时，它都会触发一个叫作 `count` 的事件：

```
<div id="app">
  <button @click="handleClick">单击了 {{ clicks }} 次</button>
</div>

<script>
  new Vue({
    el: '#app',
    data: () => ({
      clicks: 0
    }),
    methods: {
      handleClick() {
        this.clicks++;
        this.$emit('count', this.clicks);
      }
    }
```



```
  });
</script>
```

每次按钮被单击时，它都会触发 count 事件，参数为被单击的次数。

之后我们使用这个组件时，可以使用 v-on 指令来监听这个自定义事件，就和使用 v-on 指令监听 click 事件一样。下面的示例将接收 counter 组件中通过事件传递的数字并显示在页面上：

```
<div id="app">
  <counter v-on:count="handleCount"></counter>

  <p>单击次数 = {{ clicks }}</p>
</div>

<script>
  const Counter = {
    // 这里是组件的定义
  }

  new Vue({
    el: '#app',
    data: {
      clicks: 0
    },
    methods: {
      headleCount(clicks) {
        this.clicks = clicks;
      }
    },
    components: {
      Counter
    }
  });
</script>
```

在组件内部代码中，还可以使用 $on 方法来监听组件自身触发的事件。它和任何事件分发器（event dispatcher）的工作原理几乎相同：当使用 $emit 方法触发一个事件，通过 $on 方法添加的事件处理函数就会执行。不过不能使用 this.$on 方法监听子组件触发的事件；如果这么做，可以在组件上使用 v-on 指令，或者可以使用组件上的 ref 属性来调

用子组件自身的 .$on 方法：

```
<div id="app">
  <counter ref="counter"></counter>
</div>
<script>
  // 为了简单起见，该组件没有写完整
  new Vue({
    el: '#app',
    mounted() {
      this.$refs.counter.$on('count',this.handleCount);
    }
  });
</script>
```

还有另外两种处理事件的方法：$once 和 $off。$once 的行为和 $on 一样，但绑定的监听器只会执行一次——在事件第一次被触发时；而 $off 方法则用于移除一个事件监听器。这两个方法的行为与标准事件触发器（event emitter）里的方法相似，例如 Node.js 里的 EventEmitter 模块与 jQuery 中的 .on()、.once()、.off() 和 .trigger()。

由于 Vue 内置了完整的事件触发器，当你使用 Vue 时，不需要再引入自己的事件触发器了。甚至在开发 Vue 组件的局部代码时也可以利用 Vue 的事件触发器，只需要用 new Vue() 创建一个实例。请看示例：

```
const events = new Vue();

let count = 0;
function logCount() {
  count++;
  console.log(`调试函数执行了 ${count} 次`);
}

events.$on('test-event', logCount);

setInterval(() => {
  events.$emit('test-event');
}, 1000);

setTimeout(() => {
  events.$off('test-event');
}, 10000);
```


这段代码每秒会向控制台输出记录，直到 10s 后 .off() 方法将事件处理函数移除时终止。

这在处理基于 Vue 的代码与非 Vue 的代码之间的通信时，极为有用——但总的来说，只要情况允许，vuex 往往是更好的选择。

混入

混入是一种代码组织方式，可以在多个组件间横向复用代码。例如，假设你有许多用于显示不同类型用户的组件。虽然大部分显示的信息都依赖于用户的类别，但是组件间相当多的逻辑代码是共同的。有 3 种处理方式：可以为所有的组件编写重复的代码（很明显这不是一个好主意）；可以将共同的代码分离到多个函数中，并存储到 util 文件里；或者可以使用混入。后两种方式在这个例子中很相似，但是使用混入是一种更加符合 Vue 习惯的处理方式——在即将介绍的其他诸多示例中，它会更加有用。

只要将混入对象添加到组件中，那么该组件就可以获取到存储在混入对象中的任何东西。让我们创建一个混入对象，它会为所在的组件添加一个 getUserInformation() 方法：

```
const userMixin = {
  methods: {
    getUserInformation() {
      return fetch(`/api/user/${userId}`)
        .then((res) => res.json());
    }
  }
};
```

现在可以像下面这样把它添加到一个组件中并使用它：

```
import userMixin from './mixins/user';

Vue.component('usr-admin', {
  mixins: [userMixin],
  template: '<div v-if="user">姓名：{{ user.name }}</div>',
  props: {
    userId: {
      type: Number
    }
  },
  data: () => ({
```


```
    user: undefined
  }),
  mounted() {
    this.getUserInformation(this.userId)
      .then((user) => {
        this.user = user;
      });
  }
});
```

Vue 已经自动将该方法添加到组件中去了——或者说该方法被“混入”组件内了。

除了方法，混入对象可以引用几乎任何 Vue 组件所能引用的东西，就好像它是组件本身的一部分一样。例如，我们来更改示例中的混入对象，让它可以处理数据的存储和 mounted 钩子：

```
const userMixin = {
  data: () => ({
    user: undefined
  }),
  mounted(){
    fetch(`/api/user/${this.userId}`)
      .then((res) => res.json())
      .then((user) => {
        this.user = user;
      });
  }
};
```

这样组件就可以简化为：

```
import userMixin from './mixins/user';

Vue.component('user-admin', {
  mixins: [userMixin],
  template: '<div v-if="user">姓名：{{ name.user }} </div>',
  props: {
    userId: {
      type: Number
    }
  }
});
```


虽然混入使组件简化了很多，但是追踪数据的来源变得复杂了。当决定将哪些代码放在混入对象中，哪些代码放入组件中时，你必须衡量这样做的代价与收益。

混入对象和组件的合并

如果混入对象和组件间有重复的选项——比如它们都有一个叫作 addUser() 的方法或者都有一个 created() 钩子——根据它们的类型，Vue 会分别对待。

对于生命周期钩子——诸如 created() 和 beforeMount() 这样的——Vue 会将它们添加到一个数组中并全部执行：

```
const loggingMixin = {
  created() {
    console.log('mixin 中的记录 ');
  }
};

Vue.component('example-component', {
  mixins: [loggingMixin],
  created() {
    console.log(' 组件中的记录 ');
  }
});
```

当组件被创建时，“mixin 中的记录”和“组件中的记录”都将会输出到控制台。

对于重复的方法、计算属性或其他任何非生命周期钩子属性，组件中的属性会覆盖混入对象中的属性。例如：

61

```
const loggingMixin2 = {
  methods: {
    log() {
      console.log('mixin 中的记录 ');
    }
  }
};

Vue.component('example-component' {
  mixins: [loggingMixin2],
  created() {
    this.log();
```


```
  },
  methods: {
    log() {
      console.log('组件中的记录');
    }
  }
});
```

现在，当组件被创建时，只有“组件中的记录”会被输出，因为组件中的 `log()` 方法覆盖了混入中的 `log()` 方法。

有时候我们有意使用这种合并方式，但也有时候却是碰巧在不同的地方定义了相同名称的方法，如果它们中的某个方法被覆盖，可能会引发一些问题！因此，官方的 Vue 代码风格指南建议对于混入中的私有属性（不应该在混入之外使用的方法、数据和计算属性），应该在它们的名称前面添加前缀。前面的混入对象的 `log()` 方法可以写成 `$_loggingMixin2_log()`。这在开发插件时极为重要，因为用户有可能将你的插件添加到他们自己的代码中去。

vue-loader 和 .vue 文件

在本书第 4 页“安装和设置”一节中，我们讨论了如何安装 Vue，并且简单提到了如何设置 vue-loader。在编写组件时，使用 `Vue.component()` 编写组件或是将它们作为对象存储可能会有点混乱，特别是处理更加复杂的组件时，你肯定不想在组件的 template 属性中编写大量的 HTML 代码。vue-loader 提供了一种方法，可以在 .vue 文件中以有条理并且易于理解的语法编写基于单个文件的组件。

如果 vue-loader 已经设置好了，你就可以对这个前面出现过的组件进行处理：

```
Vue.component('display-number', {
  template: '<p> 当前数字是 {{ number }}</p>',
  props: {
    number: {
      type: Number,
      required: true
    }
  }
});
```


把它修改成这样：

```
<template>
  <p>当前数字是 {{ number }} </p>
</template>

<script>
  export default{
    props: {
      number: {
        type: Number,
        required: true
      }
    }
  };
</script>
```

如果将上面的代码保存为 *display-number.vue* ，之后可以将其导入到应用中并使用它，就好像你用对象语法定义了它一样：

```
<div id="app">
  <display-number></display-number>
</div>
<script>
  import DisplayNumber from './components/display-number.vue';

  new Vue({
    el: '#app',
    components: {
      DisplayNumber
    }
  });
</script>
```

经过 webpack 和 vue-loader 处理后，上面的代码的执行效果就和之前示例中的 display-number 组件一样。你必须使用预处理器，因为它无法直接在浏览器中工作。

将组件分离到文件可以让你的代码更加容易维护。可以不必将所有的组件放在同一个大文件中，而是将它们分别保存在几个文件中，并放在相关名称的目录下，同时以它们所在的网站部分来命名，或者以组件的类型或规模来命名。

63 非 Prop 属性

如果为某个组件设置的属性并不是用作 prop，该属性会被添加到组件的 HTML 根元素上。例如，假设想要给前面的 <display-number> 组件添加一个 class，你可以在引用该组件的地方为它添加 class：

```
<display-number class="some-class" :number="4"></display-number>
```

以下为输出的结果：

```
<p class="some-class">当前数字是 4</p>
```

这适用于任何 HTML 属性或特性，而不仅仅是 class。

如果我们为组件和组件的根元素设置相同的属性，会发生什么呢？大多数时候，如果我们为两者设置了相同的属性，组件上的属性会覆盖它内部模板上的属性。以下面的代码为例：

```
<div id="app">
  <custom-button type="submit">单击我！ </custom-button>
</div>

<script>
  const CustomButton = {
    template: '<button type="button"><slot></slot></button>'
  };

  new Vue({
    el: '#app',
    components: {
      CustomButton
    }
  });
</script>
```

在我们的组件模板中，将按钮设置为 type="button"，但是在引用组件时，将它设置为 type="submit"。设置在组件上的属性（而不是组件内部的属性）会覆盖掉另外的设置。以下为输出：

```
<button type="submit">单击我！ </button>
```


大部分属性都像这样，会覆盖组件内部模板中的同名属性，但是 class 和 style 稍微聪明一点，同名的值会被合并。请看下面的示例：

```
<div id="app">
  <custom-button
    class="margin-top"
    style="font-weight: bold; background-color: red">
    单击我!
  </custom-button>
</div>
<script>
  const CustomButton = {
    template: `
      <button>
        class="bustom-button"
        style="color: yellow; background-color: blue"
        <slot></slot>
      </button>`
  };

  new Vue({
    el: '#app',
    components: {
      CustomButton
    }
  });
</script>
```


class 将会被合并在一起变成 custom-button margin-top，style 属性则会被合并为 color: yellow; background-color: red; font-weight: bold;。需要注意的是，组件属性中的 background-color 样式覆盖掉了内部模板中的 background-color 样式。

组件和 v-for 指令

当使用 v-for 指令遍历一个数组或是对象，并且给定的数组或对象改变时，Vue 不会再重复生成所有的元素，而是智能地找到需要更改的元素，并且只更改这些元素。例如，如果有一个作为列表元素输出到页面上的数组，在它的末尾添加一个新元素，那么页面上现有的元素将保持不变，同时在末尾，新的元素会被创建。如果数组中间的一个元素改变了，则页面上只有对应的元素会更新。

可是，如果你在数组的中间删除或是添加一个元素，Vue 不会知道该元素对应的是页面上哪一个元素，它会更新从删除或是添加元素的位置到列表结尾之间的每一个元素。对于简单的内容，这也许不是一个问题，但对于复杂的内容和组件，你肯定不希望 Vue 这么做。

使用 v-for 指令时可以设置一个 key 属性，通过它可以告诉 Vue 数组中的每个元素应该与页面上哪个元素相关联，从而删除正确的元素。key 属性的值默认为元素在循环时的索引。可以通过下面的代码来了解这是如何工作的：

```
<template>
  <demo-key v-for="(item, i) in items" @click.native="items.splice(i, 1)">
    {{ item }}
  </demo-key>
</template>

<script>
  const randomColor = () => `hsl(${Math.floor(Math.random() * 360)}, 75%,
85%)`;
  const DemoKey = {
    template: `<p :style="{ backgroundColor: color }"><slot></slot></p>`,
    data: () => ({
      color: randomColor()
    })
  };

  export default {
    data: () => ({
      items: ['one', 'two', 'three', 'four', 'five']
    }),
    components: {
      DemoKey
    }
  };
</script>
```


click 事件的处理函数是通过 .native 修饰符添加的，因为这就是为组件添加原生 DOM 事件监听器的方式。没有 .native 修饰符，事件处理函数将不会被调用。

上面的示例会输出 5 个颜色随机的、包含数字 1 到 5 的段落元素，如下所示：

one	Mint Green
two	Pink
three	Yellow
four	Rose
five	Blue

单击其中的一个段落元素，会将对应的元素从 items 数组中删除。如果单击第二个段落元素——即图中标签为 two 的元素——希望那个元素会被彻底删除，然后标签为 three 的元素会变成第二个。实际上并不会这样！最终你得到的是下面这样的结果：

one	Mint Green
three	Pink
four	Yellow
five	Rose

这是 Vue 的差异对比机制引起的：删除了数组中的第二个元素，而原本的位置上将会变成第三个元素，于是 Vue 更新页面上对应元素的文本来反映变化，对于下个元素、下下个元素都是如此，一直到数组的结尾。最终它会删除最后一个元素。不过，这很可能并不是你想要的，所以为这个示例添加一个 key 属性，来告诉 Vue 应该删除哪个元素：

```
<template>
  <demo-key
    v-for="(item, i) in items" :key="item"
    @click.native="items.splice(i, 1)">
    {{ item }}
  </demo-key>
</template>
```

key 属性必须是一个唯一的数字或者字符串，并与数组中的元素相对应。这个例子中，

数组的元素自身就是字符串，所以可以把它们用作 key。

我经常看到有人将 key 设置为数组的下标（例如在前面例子中设置 `:key="i"`）。如果不是什么特殊情况，那么你不会想要这么做的！虽然 Vue 不会再发出警告，但你会遇到与我刚才向你展示的完全相同的问题，一个并非你期望的元素将被删除。

现在，单击第二个元素将会把 *two* 从数组中删除，同时对应的元素也将被删除，结果如下：

one	Mint Green
three	Yellow
four	Rose
five	Blue

总之，无论什么情况，只要允许就应该设置一个 key。在组件中使用 `v-for` 指令时 key 属性并不是可选的，正如你在前面的示例中看到的，在那个示例中，Vue 将会在控制台中显示一个警告。

总结

组件非常适合将代码分离到逻辑块中，并执行一个特定任务。在本章中，学习了如何创建并使用组件，既可以通过 `Vue.compoent()` 全局注册一个组件，也可以在局部使用 `components` 属性。

也看到了如何通过 prop 向组件传递数据并且验证这些属性，还有数据是如何向下传递但无法向上传递的，除非使用 `.sync` 修饰符，或者如果是 input 元素，可以使用自定义 `update` 事件。

还展示了如何使用插槽向组件传递其他 HTML 元素和组件，让你很容易就可以实现布局组件和默认内容。

看到了自定义事件，可以让你在组件中向它的父组件发送信息。

也学习了使用混入，它可以将共用的代码逻辑从组件中抽出，并在多个组件间共享。

看到了 vue-loader 并且知道了在使用 webpack 时如何使用它创建基于单个文件的组件——一个文件包含一个组件，同时清晰地分隔了 HTML 模板、JavaScript 脚本和样式代码。

最后，还了解了如果为一个组件设置特性、但不作为 prop 使用时，将会发生什么；还有在组件上使用 v-for 时为何应该使用 :key。

第 3 章

使用Vue添加样式

Vue 提供了几种方式来为网站或是应用添加样式。 v-bind:class 和 v-bind:style 两者都有专门的功能，帮助你通过数据设置 class 属性和内联样式。当结合 vue-loader 使用组件时，可以使用 scoped CSS 来添加样式，并且只在该 CSS 所在的组件中有效。

Class 绑定

通常使用 v-bind 绑定 class 属性，从而可以根据数据的变化添加或删除类名。在使用 v-bind 设置 class 属性时，Vue 增加了一些简洁的用法，使之更加易于使用。

如果使用过 React 中的 classNames 工具，那么你将会非常熟悉 v-bind 语法。v-bind 与它基本相似，不同的是 v-bind 使用一个数组将类名包裹起来，而不是将类名作为函数的参数。

如果传递一个数组给 v-bind:class，数组中的类名将会被拼接到一起。这在你想要根据数据或是计算属性设置 class 时非常方便。请看下面的示例：

```
<div id="app">
  <div v-bind:class="[firstClass, secondClass]">
    ...
  </div>
</div>
<script>
```


```
    new Vue({
      el: '#app',
      data: {
        firstClass: 'foo'
      },
      computed: {
        secondClass(){
          return 'bar';
        }
      }
    });
  </script>
```


在这个示例中，firstClass 值为 foo，secondClass 计算后得到 bar，所以 div 元素会得到一个值为 foo bar 的 class 属性。

除了可以使用数组，还可以使用对象。对象会根据条件把它的键名作为 class 添加到元素上，这取决于键名所对应的值：如果值为真，那么键名会作为 class 被添加；否则，如果是假，就不会被添加。

举个例子，假如 class 绑定的对象为 { 'my-class': shouldAddClass }，那么元素会在 shouldAddClass 执行结果为真时，拥有 my-class 这个类名。也可以在对象中指定多个 class。譬如下面的示例：

```
<div id="app">
  <div v-bind:class="classes"></div>
</div>

<script>
  new Vue({
    el: '#app',
    data: {
      shouldBeBar: true
    },
    computed: {
      classes(){
        return {
          foo: true,
          bar: this.shouldBeBar,
          hello: false
        }
```


```
      }
    }
  });
</script>
```

同样，在上面示例输出的结果中，class 为 foo bar，因为：foo 被设置为了 true，bar 被设置为了 this.shouldBeBar，执行后值也为 true，而 hello 被设置为了 false，所以没有被添加到元素上。

尽管我使用的是计算属性返回的对象，但这完全是可选的，也可以使用内联的方式设置这个对象。这只是一个很好的方式来避免代码在有很多 class 时变得太长。

当你想要同时使用变量和条件判断来添加 class 时，也可以将数组和对象混合在一起使用，只需将对象放在数组中：


```
<div v-bind:class="[
  'my-class',
  classFromVariable,
  { 'conditional-class': hasClass}
]"></div>
```

当然，如果不需要任何有关数组或是对象的功能，可以将 class 设置为一个普通的字符串或者是一个包含类名的变量。

内联样式绑定

与前面介绍的 v-bind:class 相似，Vue 也为 style 属性的设置提供了专门的功能。以前不得不通过字符串拼接来设置内联样式，现在不需要这么做了，你可以为内联样式设置一个对象：

```
<div v-bind:style="{ fontWeight: 'bold', color: 'red' }"></div>
```

注意，我们用的是 fontWeight，而不是 font-weight。Vue 会自动将该对象的属性由驼峰命名转为它们对应的 CSS 属性，这意味着不用再操心如何转义属性名中的短横杠了。

虽然使用 class 来设置元素的样式通常会更好，但是内联样式同样有用，特别是使用变量动态设置样式的时候。例如，下面这个例子会渲染出 12 种不同的颜色，形成一个颜色样板：

```
<div id="app">
  <div>
    v-for="n in 12"
    class="color"
    :style="{ backgroundColor: getColor(n) }"
  </div>
</div>
<script>
  new Vue({
    el: '#app',
    methods: {
      getColor(n){
        return `hsl(${(n-1) * 30}, 100%, 75%)`;
      }
    }
  });
</script>
```

在应用了一些 CSS 样式后，我们可以看到一个颜色样板：从一个红色的条带开始，紧接着是一个橙色的条带，一直到粉红色。

同时，Vue 会自动为你添加浏览器前缀：如果设置的某个样式需要浏览器兼容性前缀，Vue 会自动把它加上。

72 数组语法

可以使用一个数组来指定多个样式对象：

```
<div :style="[baseStyle, moreStyles]">...</div>
```

两个对象中的样式都会应用到元素上，如果有相同的样式名，那么 moreStyles 中的样式会覆盖 baseStyle 中的同名样式。

多重值

还可以使用数组提供多个值，来设置浏览器最终支持的值：

```
<div :style="{ display: ['-webkit-box','-ms-flexbox-','flex'] }"></div>
```

在上面的示例中，如果浏览器支持，结果将会是 display: flex，否则会尝试 -ms-flexbox，再是 -webkit-box。

用 vue-loader 实现 Scoped CSS

前面，你看到了如何使用 vue-loader 将组件分离到单个 .vue* 文件中。作为快速回顾，请看这个源自本书 68 页“vue-loader 与 .vue 文件”中的例子：

```
<template>
  <p>当前数字是 {{ number }}</p>
</template>

<script>
  export default {
    props: {
      number: {
        type: Number,
        required: true
      }
    }
  }
</script>
```

除了 <template> 和 <script> 标签，也可以在该文件中使用 <style> 标签，它将会在之后被输出到页面上（或者还可以设置 webpack 将样式提取到一个外部的 CSS 文件中）。假设想要为前面的例子添加样式，来让数字变为粗体：

```
<template>
  <p>当前数字是 <span class="number">{{ number }}</span></p>
</template>

<script>
  exprot default {
    props: {
      number: {
        type: Number,
        required: true
      }
    }
  };
</script>

<style>
  .number{
```


```
      font-weight: bold;
    }
  </style>
```

这样数字就变成粗体了。

与 JavaScript 不同，组件中的 CSS 不仅会影响自身，还会影响到页面上所有的 HTML 元素。Vue 有一种方式可以修复这个问题：scoped CSS。如果我们在 style 标签上添加了 scoped 特性，Vue 就会自动处理关联的 CSS 与 HTML，使编写的 CSS 只影响到该组件中的 HTML。

让我们看看它是如何工作的。如果在之前的 style 标签上添加了 scoped 属性（所以现在变成了 <style scoped>），下面是输出的 HTML：

```
<p data-v-e0e8ddca>当前数字是 <span data-v-e0e8ddca class="number">10</
span></p>

<style>
  .number[data-v-e0e8ddca]{
    font-weight: bold;
  }
</style>
```

可以看到 Vue 已经为组件中的每个元素添加了一个 data 属性，然后又将它添加到了 CSS 选择器中，使样式只应用在这些元素上。

用 vue-loader 实现 CSS Modules

作为 scoped CSS 的替代方案，可以用 vue-loader 实现 CSS Modules：

```
<template>
  <p>当前数字是 <span :class="$style.number">{{ number }}</span></p>
</template>

<style module>
  .number {
    font-weight: bold;
  }
</style>
```

74 .number 样式会被一个随机的样式名称所替换，可以通过 $style.number 来引用这个名

称，所以通过 .number 设置的样式将只会应用在那个单独的元素上。

预处理器

可以设置 vue-loader 来让预处理器处理 CSS、JavaScript 和 HTML。假设想要使用 SCSS 而不是 CSS，以利用诸如嵌套和变量这样的特性。可以通过两个步骤来实现：首先通过 npm 安装 sass-loader 和 node-sass，然后在 style 标签上添加 lang="scss"：

```
<style lang="scss" scoped>
  $color: red;

  .number {
    font-weight: bold;
    color: $color;
  }
</style>
```

就是这么简单。scoped CSS 仍会被自动处理；不再需要额外的步骤，这样就可以了。

vue-loader 还有很多其他特性，它们超出了本书的范围。如果你打算使用这些特性，我绝对推荐你查阅文档。

这些特性大部分都可以在 vueify（相当于 Browserify 版的 vue-loader）中使用。只是 vueify 并不像 vue-loader 那么流行，所以这里就不再介绍它了。

总结

在这个简短的章节中，了解到了借助 Vue 为应用添加样式的多种方式。还看到了专门的 v-bind:class 和 v-bind:style 特性，它们使添加和设置 class 与 style 变得更加简单。同时，也学到了一些 vue-loader 中与样式相关的特性：scoped CSS、CSS Modules 和预处理器。

第 4 章

render 函数和 JSX

你已经看到了如何使用 template 属性设置组件的 HTML，同样也看到了如何使用 vue-loader 在 <template> 标签中编写组件的 HTML。然而，使用模板并不是唯一能让 vue 知道应该在页面显示什么内容的方式：还可以使用 render 函数。同时，也可以在 Vue 应用中使用 JSX，如果曾经使用过 React，用起来会更加顺手（但我还是建议你尝试一下 template）。

除了使用挂载元素中的 HTML，还可以在 Vue 实例中使用 template 属性。它将会被自动添加到你指定为 Vue 挂载点的元素中。

当你将一个函数传递给 Vue 实例的 render 属性时，该函数会传入一个 createElement 函数，可以使用它来指定需要在页面上显示的 HTML。作为一个简单示例，下面代码会输出 <h1>Hello world!</h1> 到页面上：

```
new Vue({
  el: '#app',
  render(createElement){
    return createElement('h1','Hello world!');
  }
});
```

createElement 接收 3 个参数：将要生成的元素的标签名称、包含配置信息的数据对象（诸如 HTML 特性、属性、事件侦听器以及要绑定的 class 和 style 等）和一个子节点或

76 是包含子节点的数组。标签名称是必需的，另外两个是可选的(如果不需要指定数据对象，可以让子节点作为第二个参数)。让我们来逐个了解一下它们。

标签名称

标签名称是最简单的，也是唯一一个必需的参数。它可以是一个字符串，或是一个返回字符串的函数。在前面的例子中，我们的函数返回的是 h1，所以就会创建一个 <h1> 元素。render 函数中也可以访问 this，所以可以将标签名称设置为 data 对象的某个属性、prop、计算属性或是任何类似的东西，例如：

```
new Vue({
  el: '#app',
  render(createElement){
    return createElement(this.tagName, 'Hello world!');
  },
  data: {
    tagName: 'h1'
  }
});
```

这是 render 函数优于 template 的一个很大的优势，在 template 中动态设置标签名称并不是那么容易的，并且代码可读性也不好。<{{ tagName }}> 写法是无效的（而且依然不易于阅读！）。

数据对象

数据对象是设置一系列配置属性的地方，这些属性会影响生成的组件或元素。如果是在编写 template，那么它就是包括出现在标签名称与闭合尖括号 > 之间的所有东西。例如，对于 <custom-button type="submit" v-bind:text="buttonText">，相应的属性就是 type="submit" v-bind:text="buttonText"。

在这个示例中，type 是传递给组件的一个普通的 HTML 属性，而 text 是一个组件 prop，与变量 buttonText 绑定。使用 createElement 来实现相同的示例，可以这么写：

```
new Vue({
  el: '#app',
  render(createElement){
    return createElement('custom-button', {
```

```
        attrs: {
          type: 'submit'
        },
        props: {
          text: this.buttonText
        }
      });
    }
  });
```


注意到不再使用 v-bind 了，这是因为可以直接通过 this.buttonText 引用变量。由于 this.buttonText 是 render 函数的一个依赖，无论何时只要 buttonText 更新，render 函数就会被再次调用，然后 DOM 也会自动更新，就和 template 一样。

请看下面的示例，它列出了所有的选项，这些选项是实现到目前为止在本书中学习到的所有东西所必需的：

```
{
  // HTML 特性
  attrs: {
    type: 'submit'
  },

  // 传递给组件的 prop
  props: {
    text: '单击我！'
  },

  // DOM 属性，比如 innerHTML（而不是 v-html）
  domProps: {
    innerHTML: '一些 HTML'
  },

  // 事件侦听器
  on: {
    click: this.handleClick
  },

  // 与 slot="exampleSlot" 相同——当组件是某个组件的子组件时使用
  slot: 'exampleSlot',
```

```
  // 与 key="exampleKey" 相同——用于某个循环产生的组件
  key: 'exampleKey',

  // 与 ref="exampleRef" 相同
  ref: 'exampleRef',

  // 与 v-bind:class="['example-class', {'conditional-class': true}] 相同
  class: ['example-class',{ 'conditional-class': true }],

  // 与 v-bind:style="{ backgroundColor: 'red' }" 相同
  style: {backgroundColor: 'red'}
}
```

请注意，class 和 style 并没有在 attrs 属性中，它们是单独设置的。这是因为 v-bind 指令的特性；如果仅仅将 class 或者 style 设置为 attrs 对象的一个属性，就不能将 class 设置为数组或是对象，或者将 style 设置为对象。

还有一些其他的配置属性并没有在本书中涉及，请查阅官方文档以了解完整的列表。

子节点

第三个也是最后一个参数是用来设置元素的子节点的。它可以是一个数组也可以是一个字符串。如果是一个字符串，那么它的值会作为元素的文本内容被输出；如果是一个数组，可以在数组中再次调用 createElement 函数，来构建一个复杂的 DOM 树。

如果你不需要设置数据对象，只是需要配置子节点，那么可以将子节点设置为第二个参数，而不是原来的第三个参数。

以前面一节中的模板为例：

```
<div>
  <button v-on:click="count++">单击增加计数</button>
  <p>你已经单击了按钮 {{ counter }} 次。</p>
</div>
```

用 createElement 来编写同样的模板，需要像下面这样：

```
render(createElement){
  return createElement(
    'div',
    [
      createElement(
        'button',
        {
          on: {
            click: () => this.count++,
          }
        },
        '单击增加计数'
      ),
      createElement(
        'p',
        `你已经单击了按钮 ${this.counter} 次。`
      )
    ]
  );
}
```


JSX

在上面的示例中，render 函数似乎需要更多的代码来完成我们在四行模板代码中所做的工作。幸运的是，在 *babel-plugin-transform-vue-jsx* 插件的帮助下，可以使用 JSX 来编写 render 函数，Babel 插件会将 JSX 语法编译成 Vue 能理解的 createElement 函数调用形式。注意，在 Vue 内部（以及在整个 JSX 生态系统中），createElement 函数通常会有个较短的别名，h。但一般来说，不需要知道这些。

在这里并不会讲解如何安装 *babel-plugin-transform-vue-jsx*；请查看该插件的文档来了解如何安装。

在安装了 Babel 插件之后，前面的代码可以改写成这样：

```
render(){
  return (
    <div>
      <button onClick={this.clickHandler}>单击增加计数</button>
```

```
          <p>你已经单击了按钮 {counter} 次</p>
        </div>
      );
    }
```

这样就好很多了。JSX的一些其他优秀特性也可以在Vue中使用。

除了像在模板中那样，可以导入并使用组件——只要将组件的名称设置为标签的名称——还可以像 React 那样导入组件。如果导入的组件存储在首字母大写的变量中，则不需要在 components 对象中指定或是使用 Vue.component() 函数注册，就可以使用该组件。例如：

```
import MyComponent from './components/MyComponent.vue';

new Vue({
  el: '#app',
  render() {
    return (
      <MyComponent />
    );
  }
});
```

JSX的展开操作符也同样得到支持。就和 React 一样，可以在属性对象上使用展开操作符，它会与其他已经设置的属性相合并，一起应用到元素上：


```
    render() {
      const props = {
        class: 'world',
        href: 'https://www.oreilly.com/'
      };

      return (
        <a class="hello" {...props}>O'Reilly Media</a>
      );
    }
```

这会生成一个指向 oreilly.com 的链接，并且该元素的 class 属性为 hello world。

总结

本章节解释了如何使用 render 函数替代模板字符串来构建 HTML。你需要用到 createElement 函数，它接收 3 个参数：标签名称、数据对象以及元素子节点。也可以使用 JSX 替代 createElement 函数调用，这需要用到 *babel-plugin-transform-vue-jsx* 插件。

第5章

使用vue-router实现客户端路由

通过前面几章对 Vue.js 核心部分的介绍，我们已经能够在一个页面上显示并操作数据。然而，一个功能完备的网站需要的可不止这些。你可能已经在一些网站上注意到：可以在上面四处浏览，却无须从服务器上下载新的页面；又或者，之前使用过其他的框架；那么你很有可能已经和客户端路由（client-side routing）打过照面了。

vue-router 作为 Vue 的一个库，旨在让我们能够在客户端，而非服务端来处理一个应用的路由。路由就是取一个路径（如 *users/12345/ports*）来确定应该在页面上显示什么内容的行为。

安装

就 Vue 自身而言，有多种方法来安装 vue-router。可以添加如下代码来使用它的一个 CDN：

```
<script src="https://unpkg.com/vue-router"></script>
```

或者，如果用的是 npm，还可以通过 `npm install --save vue-router` 来安装它。而如果用的是诸如 webpack 的打包工具，那么就要调用 `Vue.use(VueRouter)` 来安装 vue-router：

```
import Vue from 'vue';
import VueRouter from 'vue-router';

Vue.use(VueRouter);
```

这一步往应用中添加了一些组件，你会在接下来的几节中与它们碰面。

82 基本用法

为了对该路由器进行设置，需要赋予它一组路径，以及对应的组件：路径被匹配到时，就会显示对应的组件。

下面创建一个简单的包含两个路径的路由器：

```
import PageHome from './components/pages/Home';
import PageAbout from './components/pages/About';

const router = new VueRouter({
  routes: [
    {
      path: '/',
      component: PageHome
    },
    {
      path: '/about',
      component: PageAbout
    }
  ]
});
```

现在，为了使用这个路由器，需要做两件事。首先，当应用初始化时，要把它传入到 Vue 实例中。然后，为了让它显示到页面上，需要添加一个特殊的组件 `<router-view />`。

为了把路由器传入 Vue 实例中，可以在初始化 Vue 的时候，通过 `router` 属性把路由器传进去：

```
import router from './router';
new Vue({
  el: '#app',
  router: router
});
```

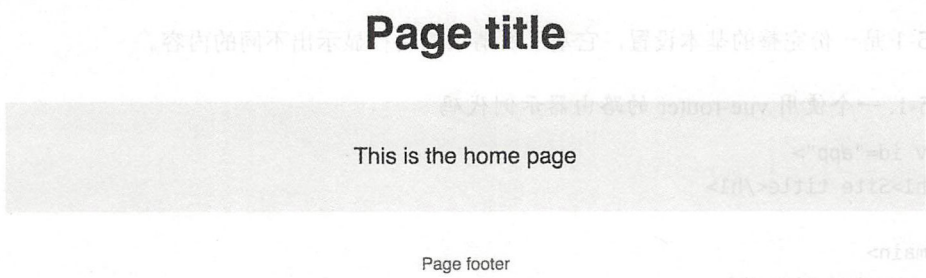
然后，在模板中，将 <router-view /> 放到任何你想让路由所返回的组件被显示的地方。

例 5-1 是一份完整的基本设置，它基于所请求的路径显示出不同的内容。

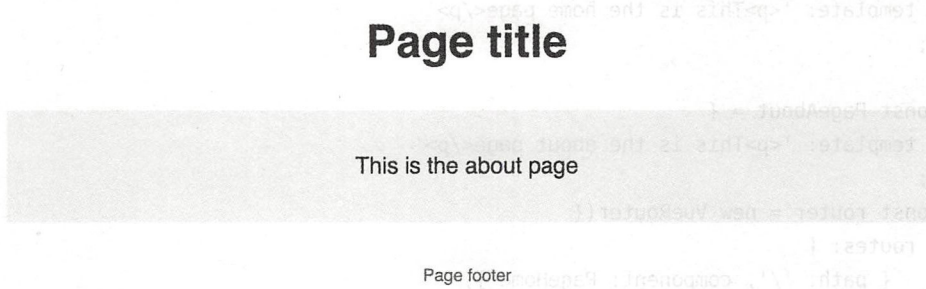
例 5-1. 一个使用 vue-router 的路由器示例代码

```
<div id="app">
  <h1>Site title</h1>

  <main>
    <router-view />
  </main>
  <p>Page footer</p>
</div>
<script>
  const PageHome = {
    template: '<p>This is the home page</p>'
  };

  const PageAbout = {
    template: '<p>This is the about page</p>'
  };
  const router = new VueRouter({
    routes: [
      { path: '/', component: PageHome },
      { path: '/about', component: PageAbout }
    ]
  });

  new Vue({
    el: '#app',
    router,
  });
</script>
```


待设置完路由器以及添加样式之后，访问根路径，会看到如下内容：



84 而访问 */about*，会看到如下内容：



HTML5 History 模式

vue-router 默认使用 URL hash 来存储路径。以前，为了访问 < 你的网站 >.com 上的 */about* 路由，你得跳转到 http://< 你的网站 >.com/#about。但如今几乎每个浏览器都已经支持 HTML5 history API，这使得开发者无须跳转到一个新的页面就能更新页面的 URL。

可以让 vue-router 使用这种 HTML5 history API，通过将路由器的 `mode` 改为 `history` 的方式：

```
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '/', component: PageHome },
    { path: '/about', component: PageAbout }
```

```
    ]
  });
```

现在，如果再跳转到 http://< 你的网站 >.com/about，你会得到……一个 404 页面。除了要告诉客户端代码去观察整个路径，而不是它的 hash，你还得告诉服务端去对每一个它不能识别的请求做出响应，并返回你所依赖的 HTML 页面（但请求诸如 CSS 等其他静态文件时则不用）。[1]

如何配置服务器取决于服务器所使用的框架，但通常来说都很简单——vue-router 文档对于几种常用服务器的配置还专门增加了一节。

动态路由

对于简单的网站而言，前面的例子已经很棒了。但如果想要做点更复杂的事情会怎样呢，比如：匹配用户的 ID 作为路径的一部分？ vue-router 支持动态路径匹配，也就是说可以通过使用一种专门的语法来指定路径规则，而所有匹配到该规则的路由下的组件都能被访问到。例如，假设我们想要让组件在路径是 */user/1234* 时被显示，而其中 *1234* 可以是任何内容，就可以像下面这样指定路由：

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      component: PageUser
    }
  ]
});
```

这样，任何匹配 */user/:userId* 的路径都会渲染出 PageUser 组件。

vue-router 使用 path-to-regexp 库来实现这项功能——同一个库也被用在 Express 和 Koa（其中官方的和非官方的路由），以及 react-router。所以如果之前有用过其中的任何一个，就不会对这种语法感到陌生。

1 许多静态网站的主机都有一个 SPA 模式，可以开启它来实现这个需求。所以先检查这个模式是否已经开启，再去考虑服务端的配置。

在组件实例中，可以通过使用属性 this.$route 来获取当前的路由对象。这个对象包括了一些有用的属性，诸如当前被访问的完整路径、URL 的查询参数（例如：?lang=en）等。就本例而言，最有用的属性莫过于 params，它包含了被动态匹配的 URL 的各个部分。在本例中，如果访问 */user/1234*，那么 params 就等于：

```
{
  "userId": "1234"
}
```

可以用这项数据去做任何你想用它去做的事。在本例中，你可能会想用它来发送一个 API 请求，以获取用户或数据，然后将它们显示到页面上。

你可能以为 userId 会是一个数值，然而 vue-router 是把它从一个路径里拽出来的（而这个路径是一个字符串），它也无从知晓它该不该是一个数值，所以 userId 实际上就保留成一个字符串。如果你想将它用作数值，就必须使用 parseFloat 或 Number 自行转换。

注意到 URL 的动态部分并不一定要在 URL 的末尾，也就是说：*/usr/1234/post* 也完全能够被匹配到。同样地，你也可以设置多段动态部分，如：*/user/1234/posts/2* 与 */user/:userId/posts/:pageNumber* 相匹配时会生成如下 params：

```
{
  "userId": "1234",
  "pageNumber": "2"
}
```


响应路由变化

当 */user/1234* 与 */user/5678* 相互切换时，其中相同的组件会被重用，于是第一章所涉及到的生命周期钩子，诸如 mounted，都不会被调用。不过，你可以使用 beforeRouteUpdate 导航守卫（guard）在 URL 动态部分变化时运行一些代码。

我们创建一个 PageUser 组件，它在挂载的时候会调用一次 API，并且在路由变化时还会再调用一次：

```
<template>
  <div v-if="state === 'loading'">
    Loading user…
```

```
    </div>
    <div>
      <h1>User: {{ userInfo.name }}</h1>
      ... etc ...
    </div>
  </template>
  <script>
    export default {
      data: () => ({
        state: 'loading',
        userInfo: undefined
      }),
      mounted() {
        this.init();
      },
      beforeRouteUpdate(to, from, next) {
        this.state = 'loading';
        this.init();
        next();
      },
      methods: {
        init() {
          fetch(`/api/user/${this.$route.params.userId}`)
            .then((res) => res.json())
            .then((data) => {
              this.userInfo = data;
        });}
      }
    };
  </script>
```


上述代码从 API 中加载了第一个用户的数据，显示其相关信息，然后如果路由发生变化（比如，用户单击了某个链接，从一个用户跳转到另一个用户），它就会再次调用 API，以获取第二个用户的数据。

注意，我们把通常位于 mounted 内部的逻辑移动到了一个方法下面（译注：即 init() 方法），在 mounted 钩子和 beforeRouteUpdate 守卫里都会调用这个方法。这就避免了重复的代码——我们几乎总是会想要在这两个地方做些相同的事。

在本章的后面会再进一步探究守卫，包括为什么要在函数的末尾调用 next()。

beforeRouteUpdate 新增于 Vue 2.2，所以它并不适用于之前的版本。在 2.2 版本之前，必须监听 $route 对象的变化：

```
const PageUser = {
  template: '<div>...user page...</div>',
  watch: {
    '$route'() {
      console.log('Route updated');
    }
  }
};
```

路由参数作为组件属性传入

除了在组件中使用 this.$route.params，还可以让 vue-router 将 params 作为路由组件的 props 传入。以如下组件为例：

```
const PageUser = {
  template: '<p>User ID: {{ $route.params.userId }}</p>'
};
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      component: PageUser
    }
  ]
});
```

当导航至 */user/1234* 的时候，“User ID: 1234”就会被输出到页面上。

88

要想让 vue-router 改为将 userId 作为组件的一个属性传入，你可以在路由中指定 props: true：

```
const PageUser = {
  props: ['userId'],
  template: '<p>User ID: {{ userId }}</p>'
};
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
```

```
        component: PageUser,
        props: true
      }
    ]
});
```

使用 `$route.$params` 代替 props 的好处是：组件与 vue-router 不再紧密耦合。设想一下，将来你想在一个页面上展示多个用户时，使用第一个例子的代码就会很棘手，但使用第二个例子的代码就会变得很简单，因为只要在另一个页面调用该组件就好了，就像它是一个与路由无关的通用组件那样。

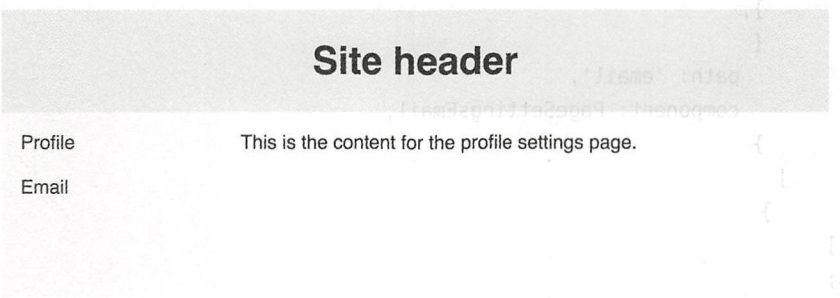
嵌套路由

在构建了一个足够复杂的应用之后，会发现在你的网站中有这么一部分，你想让其中的每一个页面都拥有普遍的样式或内容。例如，可能会有一个管理区，而你想要在网站的常规头部底下添加另一个头部，用作管理区内的导航；又比如，有一个用于展示产品的页面，而你想要添加一个页签组件，用来切换 URL 等。对于一些场景而言，比如增设管理区头部这个例子，可以很简单地将这个头部存放在组件中，然后手动引用到每一个单页上。而对于另一些场景，比如产品页面，它只有一小部分内容会变化，你肯定不想让这个独立的页签也负责展示整个页面的其余部分——此时使用嵌套路由反而会好得多。

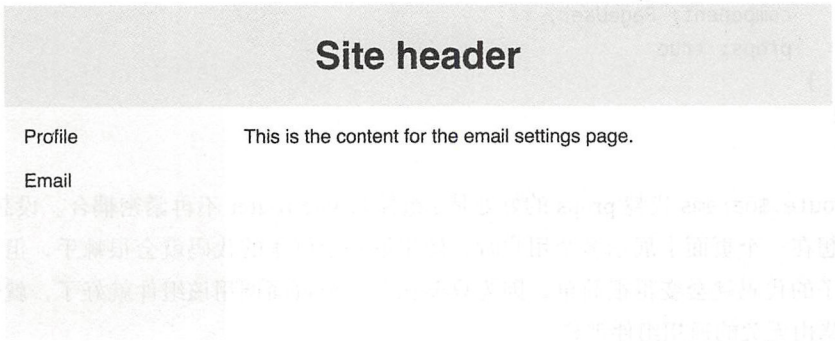
嵌套路由允许指定子路由，并且用另一个 `<router-view />` 来显示其内容。以构建一个网站的设置部分为例，它带有一个侧边栏，用于设置页面之间的导航。其中 */settings/profile* 页面用于用户修改其资料，而 */settings/email* 则允许他们修改自己的邮件偏好。

在 */settings/profile* 的用户资料页面看起来是这样的：

89



在 */settings/email* 的邮件偏好页面看起来则是这样的：



如你所见，两个页面共用了一个头部和侧边栏。头部以常规方式处理（它会位于根级`<router-view />`之外），侧边栏我们则希望它只出现在设置页面，而非所有页面。

为了达成这项需求，先创建一个 */settings* 路由，然后赋予它两个子路由：`profile`和`email`。下面就是这个路由器的配置：

```
import PageSettings from './components/pages/Settings';
import PageSettingsProfile from './components/pages/SettingsProfile';
import PageSettingsEmail from './components/pages/SettingsEmail';

const router = new VueRouter({
  routes: [
    {
      path: '/settings',
      component: PageSettings,
      children: [
        {
          path: 'profile',
          component: PageSettingsProfile,
        },
        {
          path: 'email',
          component: PageSettingsEmail,
        }
      ]
    }
  ]
});
```


接着，在`PageSettings`组件中，可以使用另一个`<router-view />`组件：

```
<div>
  <the-sidebar />
  <router-view />
</div>
```

现在，访问 */settings/profile* 会出现如下 HTML：

```
<div id="app">
  <h1>Site title</h1>

  <main>
    <div>
      <div class="sidebar">
      ...sidebar HTML...
      </div>
    </div>

    <div class="page-profile">
      ...profile settings page HTML...
    </div>
  </main>

  <p>Page footer</p>
</div>
```

可以看到，这个页面的头部和页脚位于根级 `<router-view />` 之外，而这个 `<router-view />` 就取自我们在本章第一节中所举的例子（译注：即例 5-1）；而它的侧边栏则位于设置页面内部，但在第二个 `<router-view />` 组件及其用户资料设置页面的内容之外。

重定向和别名

有些时候，比如说你已经决定要把 */settings* 重命名为 */preference*，就会想要将一个网页重定向到另一个网页。在这种情况下，你肯定不希望习惯了前往 */settings* 访问的用户看到一个错误页面，也不会希望搜索引擎链接到一堆不存在的页面上。

为了解决这个问题，可以指定一个 `redirect` 属性，用于替代 `component`：

91

```
const router = new VueRouter({
  routes: [
    {
      path: '/settings',
```


```
      redirect: '/preferences'
    }
  ]
});
```

现在，任何对 */settings* 的访问都会被重定向到 */preferences*。

另一种重定向的方法是，给组件取一个别名。比如，如果你想让设置页面从 */settings* 和 */preferences* 都可被访问，可以给 */settings* 路由取一个叫 */preferences* 的别名：

```
import PageSettings from './components/pages/Settings';
const router = new VueRouter({
  routes: [
    {
      path: '/settings',
      alias: '/preferences',
      component: PageSettings
    }
  ]
});
```

在过往的实践中发现，当我想让 */user* 和 */user/:userId* 都指向同一个组件时，路由别名就很有用。

链接导航

好啦，现在我们已经拥有一些路由了。假如此时将页面导航至 */user/1234*，你会看到，尽管服务器提供的是同一份内容，但和导航到 */preferences* 相比，还是会看到不一样的东西。那么如何让用户在不同的路由之间遨游呢？最显而易见的答案——`<a>` 标签——当然不是最好的。使用锚点来链接页面虽然在技术上行得通，但是如果你用的是 HTML5 History 模式，页面会在每次单击链接后重载，就跟在一个传统的网站里一样没区别。我们可以做得更好的嘛！鉴于现在一切都由客户端掌控，在页面之间导航是可以做到不必重载页面的，路由甚至还会替你更新 URL。

为此，使用 `<router-link>` 来代替 `<a>` 标签。它用起来与传统的锚定标签类似：

```
<router-link to="/user/1234">Go to user #1234</router-link>
```

单击该链接，它就会直接将你带往 */user/1234*，而无须加载一个新的页面。这大幅度提

升了网站性能。只要首页加载完毕，在不同页面之间的导航就会变得飞快，因为所有的 HTML、Javascript 以及 CSS 都已经被下载下来了。

除了不强制刷新页面的导航控制，vue-router 还能替你自动打理路由模式：在 hash 模式下，前面的链接会带你前往 *#/user/1234*，而使用 history 模式的话，则会带你去向 */user/1234*。

`<router-link>` 除了 `to` 以外还有一些属性，在这里列举最重要的几个，完整的属性列表需在 API 文档中查看。

tag 属性

在默认情况下，使用 `<router-link>` 会在页面上渲染出 `<a>` 标签。例如，上一节例子渲染的结果就是：

```
<a href="/user/1234">Go to user #1234</a>
```

当单击该链接时，`href` 不会起作用，因为 Vue 已经在上面添加了一个事件监听器，它取消了单击事件产生的处理自身导航的默认行为。尽管如此，出于一些原因，保留它仍然不失用处，比如在上面悬停就可以看到这个链接会跳转到哪里，或者能让你在新窗口中打开页面（这个 vue-router 就鞭长莫及了）。

不过，有些时候你可能会需要渲染除了锚点以外的元素，比如导航栏的列表元素。这时就可以使用 `tag` 属性来做到：

```
<router-link to="/user/1234" tag="li">Go to user #1234</router-link>
```

这会在页面上渲染出如下结果：

```
<li>Go to user #1234</li>
```

然后 Vue 会往这个元素添加一个事件监听器，以监测它被单击的时机，从而处理导航。

然而，这还不是最优解——因为我们就此失去了锚点标签以及它的 `href` 属性，同时也失去了几个重要的原生浏览器行为：浏览器这下就不知道这个列表项是一个链接，由此在它上面悬停也就不会给你带来任何关于这个链接的信息；也不能通过鼠标右键在新窗口打开这个链接；还有，一些诸如屏幕阅读器的辅助技术也不会把这个元素认为是一个链接了。

为了解决这个问题，可以在 <router-link> 元素里面加上锚点标签：

```
<router-link to="/user/1234" tag="li"><a>Go to user #1234</a></router-link>
```

现在，渲染出来的 HTML 就是下面这样的：

```
<li><a href="/user/1234">Go to user #1234</a></li>
```


好极了。vue-router 掌控着它力所能及的路由，而我们仍然拥有我们所期待链接能带来的常规行为。

active-class 属性

当 <router-link> 组件的 to 属性中的路径与当前页面的路径相匹配时，链接就被激活了（active）。假如处在 */user/1234*，那么该链接就是激活的。

当链接被激活时，vue-router 会自动为生成的元素赋予一个类（class）。在默认情况下，这个类是 router-link-active，不过你可以通过使用 active-class 属性来配置这个类。这很有用，特别是当你正在使用一个 UI 库，或者基于已有的代码时，而其中的链接激活类已经被设置成了其他名称。比如，你正在使用 Bootstrap 制作一个导航栏（*navbar*），就可以设置当前被激活的链接的类名为 active。

让我们用 Vue 实现一个简单的 Bootstrap 导航栏吧。以下是我们的目标：

```
<ul class="nav navbar-nav">
  <li class="active"><a href="/blog">Blog</a></li>
  <li><a href="/user/1234">User #1234</a></li>
</ul>
```

要改两个地方。第一，是 active 类要被添加到 li 元素，而不是 a 元素，于是我们要让 vue-router 渲染出一个 li 元素而不是默认的 a 元素。第二，添加的类名应该是 active，而不是 router-link-active，所以这个也要改。

下面这就是 Vue 改过的代码：

```
<ul class="nav navbar-nav">
  <router-link to="/blog" tag="li" active-class="active">
    <a>Blog</a>
  </router-link>
  <router-link to="/user/1234" tag="li" active-class="active">
    <a>User #1234</a>
```

```
    </router-link>
  </ul>
```

渲染而成的 HTML 代码与我们尚未加入 Vue 之前的例子一模一样，Vue 甚至还给每个 <a> 元素都加上了 href 属性。

原生事件

如果想给某个 <router-link> 添加一个事件处理器（event handler），可以用 @click。但是像下面这样可不行：

```
<!-- 这 样 不 行 -->
<router-link to="/blog" @click="handleClick">Blog</router-link>
```

在默认情况下，在组件上使用 v-on 就可以监听该组件触发的自定义事件，这个在第 2 章已经见过了。而对于原生事件，就可代之以 .native 修饰符来监听：

94

```
<router-link to="/blog" @click.native="handleClick">Blog</router-link>
```

现在，当单击这个链接时，handleClick 就会被调用。

编程式导航

除了通过链接的方式来为用户提供导航功能，还可以运用路由器的一些实例方法来实现编程式导航。这些方法效仿浏览器原生的 history 方法——如 history.pushState()、history.replaceState() 以及 history.go()——如果你对他们很熟悉，那么对这种写法就不会陌生。

可以用 router.push()（或组件实例中的 this.$router.push()）来进行路径跳转：

```
router.push('/user/1234');
```

这与单击一个 to 属性为 /user/1234 的 <router-link> 组件完全等效——事实上，当你单击 <router-link> 时，vue-router 内部用的就是 router.push() 方法。

还有 router.replace() 方法，它与 router.push() 的表现类似：都是将你导航至指定的路由。不同之处在于，router.push() 会向 history 栈添加一个新的记录——因此，如果用户按下返回键，路由器就会跳转到上一个路由——而 router.replace() 则替换了当前的 history 记录，所以返回键只能让你回到之前的路由。

通常情况下，你需要的都是 router.push()，但 router.repalce() 在特定情况下也不乏用处。例如，维护一个可读的文章 URL，像是 */blog/hello-world*，而一旦用户重命名了这篇文章，可能就要用 router.replace() 来继续导航至 */blog/hello-world*，因为没理由在历史记录中同时保留这两个 URL。

最后是 router.go()，它能让你在历史记录中前进和后退，就像按了前进键和后退键。后退一条记录，你就用 router.go(-1)，而前进 10 条记录，就用 router.go(10)。如果历史中没那么多条记录，函数的调用就会悄悄终止。

导航守卫

假设你想要限制未登录的用户，禁止这些用户访问你应用的某些部分。同时你还有一个 userAuthenticated() 方法用于当用户登录时返回 true。那么现在要怎么做？

95

vue-router 提供了能让你在导航发生之前运行某些代码的功能，并且遵照你的意愿去取消导航或将用户导航至其他地方。

你可以为路由器添加一个 router.beforeEach() 守卫。该守卫被传入 3 个参数：to、from 以及 next，其中 from 和 to 分别表示导航从哪里来和到哪里去，而 next 则是一个回调，在里面你可以让 vue-router 去处理导航、取消导航、重定向到其他地方或者注册一个错误。

来看一个导航守卫，它阻止未认证用户访问 */account* 路径下的任何内容：

```
router.beforeEach((to, from, next) => {
  if (to.path.startsWith('/account') && !userAuthenticated())
    next('/login');
  } else {
    next();
  }
});
```

如果被导向的路由的路径以 */account* 开头，而用户还未登录，则该用户会被重定向到 */login*；否则，就调用 next()，并且不带任何参数，然后用户就能看到他们所请求的 account 页面。有一点很重要，别忘了调用 next()，如若不然，该守卫就永远不会被解析（resolved）了！

在守卫中一个个去检查路径会让程序变得冗长且使人迷惑，特别是当你维护的是一个拥有大量路由的网站，于是会发现另一个很有用的特性，它就是路由元信息（*route meta fields*）。你可以在路由上添加一个 meta 属性，并在守卫那里重新获取它。例如，在 account 路由上设置一个 requiresAuth 属性，然后在守卫中查看该属性：

```
const router = new VueRouter({
  routes: [
    {
      path: '/account',
      component: PageAccount,
      meta: {
        requiresAuth: true
      }
    }
  ]
});

router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !userAuthenticated()) {
    next('/login');
  } else {
    next();
  }
});
```

现在，用户无论在任何时候访问 */account*，路由器都会检查其 requiresAuth 属性，如果用户尚未认证，则将其重定向登录页面。

96

当使用嵌套路由时，to.meta 指向的是子路由的元信息，而非其父路由。也就是说，如果你在 */account* 上添加了 meta 对象，而用户访问的是 */account/email*，则所获得的 meta 对象是关于该子路由的，而非父路由。可以通过遍历 to.matched 的方式来曲线救国，它同样也包含了父路由的元信息：

```
router.beforeEach((to, from, next) => {
    const requiresAuth = to.matched.some((record) => {
    return record.meta.requiresAuth;
  }
  if (requiresAuth && !userAuthenticated()) {
    next('/login');
  } else {
    next();
```

```
    }
  });
```

现在，如果访问 /account/email，而该路由又是 /account 的一个子路由，则它们两个的 meta 对象都会被检查，而不只是 /account/email 的 meta。

除了 beforeEach 这个运行在导航之前的守卫，还有一个运行在导航之后的守卫 afterEach。这个守卫只被传入两个参数，to 和 from，因此不会影响导航。但对于做些诸如设置页面标题的小事，它也不乏实用之处：

```
const router = new VueRouter({
  routes: [
    {
      path: '/blog',
      component: PageBlog,
      meta: {
        title: 'Welcome to my blog!'
      }
    }
  ]
});
router.afterEach((to) => {
  document.title = to.meta.title;
});
```

以上代码会在每次发生页面导航时观察路由的 meta 属性，并将页面的标题设置为 meta 对象的 title 属性。

97 路由独享守卫

除了在路由器上定义 beforeEach 和 afterEach 守卫，你还可以对每个单独的路由定义 beforeEnter 守卫：

```
const router = new VueRouter({
  routes: [
    {
      path: '/account',
      component: PageAccount,
      beforeEnter(to, from, next) {
        if (!userAuthenticated()) {
```

```
            next('/login');
          } else {
            next();
          }
        }
      }
    ]
});
```

beforeEnter 守卫与 beforeEach 表现完全一致，只不过这种守卫作用于每一个单独的路由而非所有。

组件内部守卫

最后，你还能在组件内部指定守卫。能使用的守卫有 3 个：beforeRouteEnter（等效于 beforeEach）、beforeRouteUpdate（你在响应路由变化一节已与它碰过面）以及 beforeRouteLeave（在导航离开一个路由时调用）。所有这 3 个守卫都接受 3 个与 beforeEach 和 beforeEnter 一样的参数。

让我们回到前面那个认证的例子，并将相同的逻辑套到组件里：

```
const PageAccount = {
  template: '<div>...account page...</div>',
  beforeRouteEnter(to, from, next) {
    if (!userAuthenticated()) {
      next('/login');
       } else {margin-bottom
      next();
    }
  }
};
```

注意，在 beforeRouteEnter 中 this 是 undefined，因为此时组件还尚未被创建。但是，可以在 next 里传一个回调，该回调会被传入组件实例并作为其第一个参数：

```
const PageAccount = {
  template: '<div>...account page...</div>'
  beforeRouteEnter(to, from, next) {
    next((vm) => {
        console.log(vm.$route);
    });
```

```
  }
};
```

鉴于你在 beforeRouteUpdate 和 beforeRouteLeave 里都能使用 this，就像在组件内的其他大部分地方一样，因此这两个守卫并不支持在 next 里传入一个回调。

路由顺序

vue-router 在内部通过遍历路由数组的方式来挑选被显示的路由，并选取其中匹配到当前 URL 的第一个。知晓这点很重要——这意味着安排好路由的顺序很重要。以如下路由器为例：

```
const routerA = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      component: PageUser
    },
    {
      path: '/user/me',
      component: PageMe
    }
  ]
});

const routerB = new VueRouter({
  routes: [
    {
      path: '/user/me',
      component: PageMe
    },
    {
      path: '/user/:userId',
      component: PageUser
    }
  ]
});
```

这两个路由器看起来很像：它们都定义了两个路由，一个让用户访问他们自己的页面 */user/me*，而另一个用于访问其他用户的页面 */user/:userId*。

但是使用 routerA 的话，是不可能访问到 PageMe 的。这是因为使用 A 路由时，vue-router 查看了整个路由器，并测试哪个路径规则与之相匹配，即如果访问 */user/me* ，匹配的就是 */user/:userId* ，所以 userId 就被设置成了 me，从而显示的就是 PageUser 组件。而对于 B 路由，先匹配到的是 */user/me* ，所以其对应组件就被启用了，于是任何由 PageUser 生成的内容也被相应显示。

404 页面

可以利用 vue-router 会按顺序搜索路由直到与通配符 (*) 匹配的特点，来渲染一个显示错误页面。像下面这么简单做即可：

```
const router = new VueRoute({
  routes: [
    // ... 你的其他路由 ...
    {
      path: '*',
      component: PageNotFound
    }
  ]
});
```

当其他路由都匹配不到时，就会显示 PageNotFound 组件。

在使用嵌套路由时，如果没有匹配到子路由，则路由器会继续往下对其父路由之外的路由列表进行搜寻，所以通配符路由返回的都是同样的 PageNotFound 组件。如果想让子路由的错误页面也能在父组件中显示，则需要在子路由数组中添加该通配符路由：

```
const router = new VueRoute({
  routes: [
    {
      path: '/settings',
      component: PageSettings,
      children: [
        {
          path: 'profile',
          component: PageSettingsProfile
        },
        {
          path: '*',
          component: PageNotFound
```

```
        }
      ]
    },
     {
      path: '*',
      component: PageNotFound
    }
  ]
});
```

100 现在显示的还是同一个 PageNotFound 组件，只不过单击 PageSettings 组件的侧边栏也能显示这个组件。

路由命名

在本章要说的最后一个 vue-router 的特性是其命名路由的能力，然后用路由的名称来代替它们的路径。先给路由器的路由加上一个 name 属性：

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      name: 'home',
      component: PageHome
    },
    {
      path: '/user/:userId',
      name: 'user',
      component: PageUser
    }
  ]
});
```

现在，可以使用名称来代替路径进行链接了：

```
<router-link :to="{ name: 'home' }">Return to home</router-link>
```

下面两行代码是等效的：

```
<router-link to="/user/1234">User #1234</router-link>
<router-link :to="{ path: '/user/1234' }">
```

```
    User #1234
  </router-link>
```

只不过第一行比第二行要短。

还可以用 router.push() 来跳转：

```
router.push({ name: 'home' });
```

给路由命名，并采取名称而不是路径来标识它们，这意味着路由和路径变得不再那么耦合紧密了：改变路由的路径时，只需要改变路由器中的路径，而不需要再去检查所有现存的链接并对它们做出更新。

要想像之前的 user 路由那样把路由与 params 关联起来，你可以在 to 对象的 params 属性中指定：

```
<router-link :to="{ name: 'user', params: { userId: 1234 }}">
  User #1234
</router-link>
```

101

总结

在本章中，看到了如何使用 vue-router 构建一个单页应用——它带有多个页面，其路由受客户端掌控。也看到了配置路由器的各种方法，并且使用动态路由创建动态路径，还有使用嵌套路由创建子路由以及重定向、别名、404 页面的通配路径等。还看到了用 <router-link> 来创建链接、用路由命名来解耦路由对象和路由路径以及用导航守卫来在导航事件发生时执行附加逻辑。

第 6 章

使用 vuex 实现状态管理

书说至此，我们编写的组件的所有数据都是存放在组件内部的。调用一个 API，然后把返回的数据存放在一个数据对象中。把一个表单绑定到一个对象上，还是把这个对象存放在这个数据对象中。组件之间的所有通信都采用事件（events）方式（子组件往父组件通信）和属性（props）方式（父组件往子组件通信）。在简单的应用场合中，这一套用着不错，但在稍微复杂一点的应用中，就捉襟见肘了。

用一个社交网络应用来举个例，就说其中的消息部分。比如说，你想在应用顶部导航栏上放一个图标，用来显示收到的消息数量，同时在页面底部，还想要一个消息弹窗，同样是告诉你收到的消息数量。因为图标和弹窗这两个组件彼此在页面上并无直接联系，所以用 events 和 props 来连接它们将会是一场噩梦：与消息通知无关的组件将不得不传递这些额外的事件（译注：因为那两个与消息通知有关的组件之间没有直接联系，并非父子关系，如果用事件方式或属性方式通信，则必然要经过其他组件传递事件或属性）。另外一种方法是，不通过连接两个组件的方式来共享数据，而是每个组件各自发送 API 请求。但这么做会更糟：不同的组件将会在不同的时间点更新，这就意味着它们会渲染不一样的数据，并且页面所发送的 API 请求也会远远超过其实际所需。

vuex 应运而生，帮助开发者管理 Vue 应用中的状态。它提供了一种集中式存储（centralized store），可以在整个应用中使用它来存储和维护全局状态。它同时还使你能够对存入的数据进行校验，以保证当这个数据再次被取出时是可预见而且正确的。

安装

可以用 CDN 来引入 vuex，只需加入如下代码：

```
<script src="https://unpkg.com/vuex"></script>
```

104 此外，如果使用的是 npm，也可以通过 `npm install --save vuex` 来安装 vuex。而如果使用的是诸如 webpack 的打包工具，那么就要像使用 vue-router 那样，调用 `Vue.use()`：

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);
```

然后就是创建 store。创建并保存 *store/index.js* 文件，内容如下：

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {}
});
```

现在这个 store 还空无一物，这一整章都会往里面加东西。

接着，将它引入主应用文件中，并在 Vue 实例化时作为一个属性传入。

```
import Vue from 'vue';
import store from './store';

new Vue({
  el: '#app',
  store,
  components: {
    App
  }
});
```

现在，已经把这个 store 引入应用中了，并且可以用 `this.$store` 来访问它。

接下来先看看有关 vuex 的一些概念，然后再来看看能用 `this.$store` 做些什么。

概念

正如本章开头所提到的，vuex 可以满足复杂应用中多个组件进行状态共享的需求。

用一个简单的组件来说明一下，这个组件显示的是用户在页面上所拥有的消息数目，它不使用 vuex。

```
const NotificationCount = {
  template: `<p>Messages: {{ messageCount }}</p>`,
  data: () => ({
    messageCount: 'loading'
  }),
  mounted() {
    const ws = new WebSocket('/api/messages');

    ws.addEventListener('message', (e) => {
      const data = JSON.parse(e.data);
      this.messageCount = data.messages.length;
    });
  }
};
```

105

这个组件相当简单。它打开了一个通往 */api/message* 这个地址的 websocket 通道，然后当服务器发送数据给客户端时——在本例中，就是当 socket 打开时（发送消息的初始数目），以及当这个数目更新时（也就是有新消息时）——在这个 socket 上发送的消息数量就会被显示在页面上。

在实际应用中，这份代码会更复杂。因为在这个例子中并没有进行 websocket 认证，并且假设了在 websocket 上获得的服务器响应总是一个合法的 JSON，同时这个 JSON 带有一个 `messages` 属性，并且这个 `messages` 属性是一个数组。实际情况可能与此相悖，但就例子而言，使用简化的代码来完成工作。

当我们想要在同一个页面上使用不止一个 `NotificationCount` 组件时，问题就来了。因为每个组件都会打开一个 websocket 通道，所以就创建了一些不必要的重复连接，并且由于网络延迟，各个组件的更新时间可能会稍稍有点不同。为了解决这个问题，可以把 websocket 的逻辑放入 vuex 中。

立即通过一个例子来深入了解。修改一下之前的组件：

```
const NotificationCount = {
  template: `<p>Messages: {{ messageCount }}</p>`,
  computed: {
    messageCount() {
      return this.$store.state.messages.length;
    }
  }
  mounted() {
    this.$store.dispatch('getMessages');
  }
};
```

然后下面是我们的 vuex store（store/index.js）：

```
let ws;

export default new Vuex.Store({
  state: {
    messages: [],
  },
  mutations: {
    setMessages(state, messages) {
      state.messages = messages;
    }
  },
  actions: {
    getMessages({ commit }) {
      if (ws) {
        return;
      }

      ws = new WebSocket('/api/messages');

      ws.addEventListener('message', (e) => {
        const data = JSON.parse(e.data);
        commit('setMessages', data.messages);
      });
    }
  }
});
```

现在，虽然每个已挂载的通知计数组件都会触发 getMessages 动作，但是这个动作会检查是否存在 websocket 实例，并且只在无连接的时候创建连接。连接后，它就开始监听这个连接套接字（socket），提交状态变更，这个变更会在所有的通知计数组件上得以反映，这是因为 store 是响应式的——正如 Vue 中大多数的其他事物那样。当这个套接字收到新的数据时，这个全局 store 就会跟着更新，同时这个页面上的每个组件也都会跟着更新。

在本章的剩余部分，引入一些单独的概念，并把它们加入例子当中，它们就是 state（状态）、mutation（变更）和 action（动作），同时还会阐述大型应用中构建 vuex 模块的方法，以免这些模块形成一个过分庞大而且混乱的文件。

State 及其辅助函数

首先来看看 state。state 表示数据在 vuex 中的存储状态，它就像一个在应用的任何角落都能访问到的庞大对象——是的，它就是*单一数据源*（*single source of truth*）。

来看看仅存储了一个数字的 store:

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {
    messageCount: 10
  }
});
```

这里可以通过 `this.$store.state.messageCount` 来访问应用中 state 对象的 messageCount 属性。不过这么做有点儿烦琐，通常把它放入一个计算属性中会更好，像这样：

107

```
const NotificationCount = {
  template: `<p>Messages: {{ messageCount }}</p>`,
  computed: {
    messageCount() {
      return this.$store.state.messageCount;
    }
  }
};
```


现在，组件就会显示“Message: 10”了。

State 辅助函数

当引用 store 的属性不多时，直接在计算属性中访问 store 是没问题的，但是当你引用大量的 store 属性时，多次引用就会变得很重复。鉴于此，vuex 提供了一个辅助函数 mapState，它返回一个被用作计算属性的函数对象。

通常情况下，只需要提供一个字符串数组：

```
import { mapState } from 'vuex';

const NotificationCount = {
  template: `<p>Messages: {{ messageCount }}</p>`,
  computed: mapState(['messageCount'])
};
```

这段代码所做的和之前的示例代码一样，只是更短一些。如果我们需要从 vuex store 中获取更多的属性，就可以简单地在 mapState 的调用中加入这些属性的名称。

上面 mapState 的调用其实是下面的缩写：

```
computed: mapState({
  messageCount: (state) => state.messageCount
})
```

mapState 函数以一个对象作为参数，并将其中的各个键值分别映射到一个计算属性。如果键值给定的是函数，则该函数会以 state 作为其第一个参数被调用，从而使你能够从这个参数上获取 state 的值。如果只是要从 state 上获取一个属性作为计算属性，也可以只赋予它一个字符串：

```
computed: mapState({
  messageCount: 'messageCount'
})
```


到此，已见识过 3 种方法，它们从 store 中获取完全相同的值，并将这个值用作完全相同的计算属性，你可能会好奇这 3 种方法有何区别。

如果所做的所有映射都只是简单地将计算属性的名称映射到 vuex 中的属性名——比如，messageCount 映射到 state.messageCount——那么可以使用第一个例子所展示的那种数

组写法。

但是如果属性中有要映射到不同名称的，或者需要进行处理时，就需要用完整对象写法。建议你尽量使用数组写法——它阅读起来十分简单——而在需要进行处理时则使用函数的写法。例如，state 拥有一个 message 属性，它存储着一个消息数组，而我们需要这个消息数组的长度作为 messageCount，则此时可以编写如下代码：

```
computed: mapState({
  messageCount: (state) => state.messages.length,
  somethingElse: 'somethingElse'
})
```

当然了，使用哪种写法都由你来定。

如果使用的是完整函数写法，即非 ES6 箭头函数的写法，还可以通过 this 来访问组件实例，从而混合使用本地 state 和 vuex 全局 state：

```
computed: mapState({
messageCount(state) {
  return state.messages.length + this.pendingMessages.length;
}
})
```

在上面这个例子中，我们混合使用了 vuex 全局的 state 和组件本地的 state。此时使用者会犯的一个常见错误，是将其应用的所有 state 都放入 vuex，而这并非所愿。使用组件本地的 state 没有问题，但最好能结合这两条建议：应用级的 state 放入 vuex 中以便组件之间共享，而只用于单个组件内部的简单 state 则作为组件本地 state。

最后，来看一种能将 mapState 结合到已有的计算属性的方法。因为 mapState 返回的是一个对象，所以如果已经拥有计算属性，那么就需要一种方法来将这两个对象合并到一起。幸运的是，stage-3 ECMAScript 已有了提案（即纳入 JavaScript 的候选方案），其中的对象展开符在这里可以帮到我们。它就像 ECMAScript 2015 中的数组展开符一样，只是作用于对象：

```
computed: {
  doubleFoo() {
    return this.foo * 2;
  },
```


```
    ...mapState({
      messageCount: (state) => state.messages.length,
      somethingElse: 'somethingElse'
    })
  }
```

对象的展开结果为：

```
  computed: {
    doubleFoo() {
      return this.foo * 2;
    },
    messageCount() {
      return this.$store.state.messages.length,
    },
    somethingElse() {
      return this.$store.state.somethingElse;
    }
  }
```

注意，如果使用了对象展开符，那么可能会需要像 Babel 这样的编译工具来确保最大限度的浏览器支持。

Getter

有时候，你可能会发现在多个组件中使用着相同的数据，并对它们做着相同的处理，这产生了很多重复的代码。例如，我们可能会发现在重复计算那些消息未被读取的用户的名称：

```
  computed: mapState({
    unreadFrom: (state) => state
      .filter((message) => !message.read)
      .map((message) => message.user.name)
  })
```

如果只是使用一次这个计算属性那还好，要是在多个组件当中都用到它，那重复的劳动可就大了。此外，如果 API 返回数据格式发生了变化（message.user.name 可能要改成 message.sender.full_name），到时就不得不更新很多位置的代码。

幸运的是，vuxe 为我们提供了 getters 属性，它使我们能够将通常会被重复使用的代码

移动到 vuex store 内部，从而避免产生冗余。

来看看怎么把上面那个例子中的重复代码移动到 vuex store：

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {
    messages: [...]
  },
  getters: {
    unreadFrom: (state) => state.messages
      .filter((message) => !message.read)
      .map((message) => message.user.name)
  }
});
```


现在，unreadFrom 这个 getter 即可通过 store.getters.unreadFrom 来访问：

```
computed: {
  unreadFrom() {
    return this.$store.getters.unreadFrom;
  }
}
```

优雅。另外，通过使用 getter 的第二参数，不同 getter 之间可以互相访问。例如，把上面的 getter 拆分成两个小 getter：

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {
    messages: [...]
  },
  getters: {
    unread: (state) => state.filter((message) => !message.read),
    unreadFrom: (state, getters) => getters.unread
      .map((message) => message.user.name)
  }
});
```


Getter 辅助函数

和 state 一样，getter 方法也有辅助函数，省去了每次都要调用 this.$store.getters。getter 辅助函数和 mapState 用起来类似，不过不支持函数写法。

它有数组写法：

```
computed: mapGetters(['unread', 'unreadFrom'])
```

等效于下面的写法：

```
computed: {
  unread() {
    return this.$store.getters.unread;
  },
  unreadFrom() {
    return this.$store.getters.unreadFrom;
  },
}
```

111

还有对象写法：

```
computed: mapGetters({
  unreadMessages: 'unread',
  unreadMessagesFrom: 'unreadFrom'
})
```

等效于下面的写法：

```
computed: {
  unreadMessages() {
    return this.$store.getters.unread;
  },
  unreadMessagesFrom() {
    return this.$store.getters.unreadFrom;
  },
}
```

Mutation

到目前为止，只见过如何将数据从 store 中取出，却还没看到如何对数据进行修改。不可以直接修改 state 对象的值，得使用 mutation。

mutation 是一个函数，它对 state 进行同步变更，通过调用 `store.commit()` 并传入 mutation 名称的方式来达成。

现在创建一个 mutation，用于在消息数组的尾部追加一条消息：

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {
    messages: []
  },
  mutations: {
    addMessage(state, newMessage) {
      state.messages.push(newMessage);
    }
  }
});
```

然后，在组件中调用 `store.commit()` 来追加一条信息：

```
const SendMessage = {
  template: '<form @submit="handleSubmit">...</form>',
  data: () => ({
    formData: { ... }
  }),
  methods: {
    handleSubmit() {
      this.$store.commit('addMessage', this.formData);
    }
  }
};
```

112

`commit` 方法的第一个参数即为 mutation 的名称，而第二个则是一个可选参数，即 `payload`。

这个 mutation 方法同样支持对象写法：

```
this.$store.commit({
  type: 'addMessage',
  newMessage: this.formData
})
```


这样整个对象都是 payload，所以该 mutation 往消息数组追加的应该是 `payload.newMessage` 而不是整个 payload。

Mutation 辅助函数

与 state 和 getters 一样，mutation 也有 `mapMutations` 方法。它与辅助函数 `mapGetters` 拥有完全相同的写法。

它有数组写法：

```
methods: mapMutations(['addMessage'])
```

等效于下面的写法：

```
methods: {
  addMessage(payload) {
    return this.$store.commit('addMessage', payload);
  },
}
```

正如你看到的，它也支持 payload 参数（再重申一下，这个参数是可选的）。

另外，还有对象写法，如果想让 mutation 的方法名称不同于 mutation 的名称的话：

```
methods: mapMutations({
  addNewMessage: 'addMessage'
})
```

等效于下面的写法：

```
methods: {
  addNewMessage(payload) {
    return this.$store.commit('addMessage', payload);
  },
}
```

113

Mutation 必须是同步函数

正如前面所提到的，mutation 只能实现同步变更 state 对象。如果需要实现异步变更，那么应该使用 *action*。

Action

终于，来到了 action 部分。用 mutation 只能做到同步变更，而 action 则用于实现异步变更。

我们先实现一个 action，它基于之前所看到的 state 和 mutation。这个 action 会使用 `fetch` API 来向服务器发送请求，以检查是否有新的消息，然后再把新消息追加到 `message` 数组的末尾：

```
import Vuex from 'vuex';

export default new Vuex.Store({
  state: {
    messages: []
  },
  mutations: {
    addMessage(state, newMessage) {
      state.messages.push(newMessage);
    },
    addMessages(state, newMessages) {
      state.messages.push(...newMessages);
    }
  },
  actions: {
    getMessages(context) {
      fetch('/api/new-messages')
        .then((res) => res.json())
        .then((data) => {
          if (data.messages.length) {
            context.commit('addMessages', data.messages);
          }
        });
    }
  }
});
```

这个例子中有两处新事物：首先是一个名为 `addMessages` 的 mutation，它与 `addMessage` 类似，不过可以一次性追加多条消息；然后是名为 `getMessages` 的 action，它检查服务器上的新消息，并且如果存在，则调用 `addMessages` 并传入这些新消息。

然后，使用 `store.dispatch()` 方法来在组件中调用 `getMessages`。可以为

NotificationCount 组件添加一个链接，以更新消息的数目：

114

```
import { mapState } from 'vuex';

const NotificationCount = {
  template: `<p>
    Messages: {{ messages.length }}
    <a @click.prevent="handleUpdate">(update)</a>
  </p>`,
  computed: mapState(['messages']),
  methods: {
    handleUpdate() {
      this.$store.dispatch('getMessages');
    }
  }
};
```

现在，单击链接将触发如下流程：

1. 链接被单击，触发 handleUpdate 方法。
2. 相应的 getMessages 被分发（dispatch）。
3. 相关的请求被送往 */api/new-messages*。
4. 请求获得响应时，如果存在新消息，则调用 addMessages 并将这些新消息作为其 payload 参数。
5. addMessages 将这些新消息追加到 state 的 messages 属性中。
6. 由于 state 发生了变化，计算属性 messages 就会更新，从而页面上的文本显示也会相应改变。

dispatch 调用语法与 commit 相同：它既可以接受 action 的名称作为其第一个参数、payload 对象作为其第二个参数，也可以接受一个包含 type 属性的对象作为其 payload 参数。

Action 辅助函数

同理，action 也存在一个 mapActions 辅助函数，用于将普通方法映射到 action：

```
methods: {
  // 将 this.getMessage() 映射到 this.$store.dispatch('getMessage')
  ...mapActions(['getMessage'])

  // 将 this.update() 映射到 this.$store.dispatch('getMessages')
  ...mapActions({
    update: 'getMessages'
  })
}
```

参数解构

在 action 中使用参数解构是一种相当标准的做法，以代替对 context 对象的引用，像这样：

115

```
actions: {
  getMessages({ commit }) {
    // 做些事情，然后……
    commit('addMessages', data.messages);
  }
}
```

context 对象即等于 vuex 的 store——或者说是下一节将要一探究竟的——当前的 vuex 模块。通过 context 对象，可以访问到 state(即它的 state 属性)，不过不能直接修改它——还是得通过提交 mutation 来变更。

Promise 与 Action

action 是异步函数，我们怎么知道它们已经完成了呢？可以观察计算属性的改变，但这不够理想。

其实可以在 action 中返回一个 promise 对象来代替上述做法。另外，调用 dispatch 也会返回一个 promise 对象，运用它就可以在 action 运行结束时去运行其他代码。

给前面的 getMessages 做些改变，让它返回一个 promise 对象：

```
actions: {
  getMessages({ commit }) {
    return fetch('/api/new-messages')
      .then((res) => res.json())
      .then((data) => {
```

```
        if (data.messages.length) {
          commit('addMessages', data.messages);
        }
      });
  }
}
```

改动很小——只是在调用 `fetch` 之前加了一个 `return`。现在，我们改变一下我们的 `NotificationCount` 组件，让它在消息数目更新的过程中显示一个正在加载的标志。为了做到这一点，需要往组件本地的 state 中添加一个 `update` 属性，非 `true` 即 `false`：

```
import { mapState } from 'vuex';

const NotificationCount = {
  template: `<p>
              Messages: {{ messages.length }}
              <span v-if="loading">(updating…)</span>
              <a v-else @click.prevent="handleUpdate">(update)</a>
            </p>`,
  data: () => ({
    updating: false,
  }),
  computed: mapState(['messages']),
  methods: {
    handleUpdate() {
      this.updating = true;
      this.$store.dispatch('getMessages')
        .then(() => {
          this.updating = false;
        });
    }
  }
};
```

116

现在，当组件更新消息数目时，就会显示 (updating…)，而不是一个更新消息数目的链接。

Module

鉴于你已经知道了如何在 vuex store 中存储数据，也知道了如何修改数据，现在来谈谈如何组织 store。

在较小的应用中，目前为止你所看到的组织方法——即在单一的文件中维护应用中所有的 state、getter、mutation 以及 action——尚且行之有效。但是在较大型的应用中，就会显得有点杂乱，因此 vuex 允许你将你的 store 拆分到各个模块（module）中。

每个 *module* 都只是一个对象，并且拥有其自身的 state、getter、mutation 以及 action，通过使用 modules 属性即可将它们添加到 store 当中。以之前的 store 为例，将其中的消息处理逻辑（对，全部）分离到一个模块中：

```
import Vuex from 'vuex';

const messages = {
  state: {
    messages: []
  },
  mutations: {
    addMessage(state, newMessage) {
      state.messages.push(newMessage);
    },
    addMessages(state, newMessages) {
      state.messages.push(...newMessages);
    }
  },
  actions: {
    return getMessages({ commit }) {
      fetch('/api/new-messages')
        .then((res) => res.json())
        .then((data) => {
          if (data.messages.length) {
            commit('addMessages', data.messages);
          }
        });
    }
  }
};

export default new Vuex.Store({
  modules: {
    messages,
  }
});
```

117

做完这项变动之后，这个 store 现在的运行方式就发生了一些微妙的变化。

首先一个变化是 mutation 和 getter 中的 state 现在指向的则是该模块的 state，而非根节点的 state（即主 store 的 state），并且 action 当中的 context 对象指向的也是 module 而非 store。在该模块内部所做的处理，都只影响该模块，而不会影响到其他任何模块。在 getter 中，可以通过其第三个参数 rootState 属性来访问根节点状态，而在 action 中，则可以通过 context 对象的 rootState 属性来访问。

第二个变化，是当你从 store 中取数据时必须指定从哪个模块取。即应该访问 state.messages.messages（而非 state.messages），其中第一个 messages 为模块名称，而第二个 messages 才是所需状态的值。

在本例中无须考虑第一个变化——至少一切尚且如期运转——而第二个变化则要求我们在 Notification 组件中对 mapState 的调用做一点改变。我们可以将模块名称作为 mapState 的第一个参数：

```
computed: mapState('messages', ['messages'])
```

这意味着如果你要从多个模块中获取 state，将不得不多次调用 mapState，不过有了对象展开符，问题不大。

文件结构

通常，我会乐于将各个模块拆分到其各自的文件中。这会让代码变得更优雅也更有组织性，而且还可以用 ES6 的 export 语法让模块做到真正的简洁。因此我们把关于消息处理的代码移入一个模块文件，并保存为 *store/modules/messages.js*：

```
export const state = {
  messages: []
};

export const mutations = {
  addMessage(state, newMessage) {
    state.messages.push(newMessage);
  },
  addMessages(state, newMessages) {
    state.messages.push(...newMessages);
  }
};
```

```
export const actions = {
  return getMessages({ commit }) {
    fetch('/api/new-messages')
      .then((res) => res.json())
      .then((data) => {
        if (data.messages.length) {
          commit('addMessages', data.messages);
        }
      });
  }
};
```

然后就可以在 *store/index.js* 当中导入这个模块：

```
import Vuex from 'vuex';
import * as messages from './modules/messages';

export default new Vuex.Store({
  modules: {
    messages,
  }
});
```

当然了，也可以将该模块整个作为一个对象并以默认方式输出（官方文档做法即是如此），我觉得这种做法会更好一些。

带命名空间的模块

在默认情况下，只有 state 是带命名空间的。而模块内部的 getter、mutation 和 action 仍然与未做模块拆分时的调用方式完全相同，而且如果被分发的 action 同时存在于多个模块中时，则每个模块中的该 action 都会被分发。这种特性有其用处，但也是个潜在的隐患。而为整个模块创建命名空间的做法则有可能避免这种隐患的发生。

要让 vuex 为模块创建命名空间，需要在模块对象中加入 `namespaced：true`——在之前所举的例子的模块文件中，即添加如下代码：

```
export const namespaced = true;
```

现在要访问 getter，则应在其名称之前指定命名空间的名称：

119

```
  computed: {
    unreadFrom() {
      return this.$store.getters['messages/unreadFrom'];
    }
  }
```

而触发 mutation 和分发 action 时，也是一样的做法——在其名称之前指定命名空间名称：

```
store.commit('messages/addMessage', newMessage);

store.dispatch('messages/getMessages');
```

同 mapState 一样，其他 3 个辅助函数——mapGetters、mapMutations 和 mapActions——也接受带命名空间的模块名称作为其第一个参数。

来看看模块命名空间对 NotificationCount 组件产生了哪些影响：

```
import { mapState } from 'vuex';

const NotificationCount = {
  template: `<p>
    Messages: {{ messages.length }}
    <span v-if="loading">(updating…)</span>
    <a v-else @click.prevent="handleUpdate">(update)</a>
    </p>`,
  data: () => ({
    updating: false,
  }),
  computed: mapState('messages', ['messages']),
  methods: {
    handleUpdate() {
      this.updating = true;
      this.$store.dispatch('messages/getMessages')
        .then(() => {
          this.updating = false;
        });
    }
  }
};
```

没有太大的改动，但很明显，模块已经拥有自己的命名空间了。

总结

本章，我们介绍了使用 vuex 来管理复杂应用中的状态，以及 vuex 中各种各样的概念：

- vuex store 是一切事物——state、getter、mutation 和 action——被存储和访问的必由之所。
- state 是应用中所有数据存放的对象。
- getter 使你能够将通用的逻辑聚合起来，以获取 store 中的数据。
- mutation 用于同步变更 store 中的数据。
- action 用于异步变更 store 中的数据。

120

state、getter、mutation 和 action 都有其各自的辅助函数，用于协助你将它们加入组件当中，它们分别是 `mapState`、`mapGetters`、`mapMutations` 和 `mapActions`。

最后，还看到了使用模块来切分 vuex store，使其成为一个个包含各自逻辑的代码块。

第 7 章

对Vue组件进行测试

通过将代码划分为多个组件，为应用程序编写单元测试将会变得十分容易。现在代码已经被划分为多个小块，每个小块仅负责单个功能，每个功能只有相对较少的配置项——这些都十分有利于测试。为代码增加单元测试，可以确保在未来对代码做出更改时，不会破坏那些你不希望发生改变的功能。只要编写的应用程序具备一定的规模，你就可能应该为它编写一些单元测试。

在这一章中，将首先看看在 Vue 中，如何不使用任何库对组件进行单元测试，然后再通过 vue-test-utils 这个库来使测试代码的语法变得更加优雅和简洁。

在本书中，我们不会在测试框架（如 Jasmine、Mocha 或者 Jest）上花费太多时间。我们所编写的大多数测试代码会直接抛出可在浏览器控制台被检查的错误。如果想更深入地了解 JavaScript 中有关于测试的知识，可以阅读由 Evan Hahn 编写的书籍 *JavaScript Testing with Jasmine* 或者其他在线文章。

测试单个组件

首先从一个能被测试的简单组件开始，这个组件是之前章节中 NotificationCount 组件的简化版本：

```
const NotificationCount = {
  template: `<p>
      Messages: <span class="count">{{ messageCount }}</span>
      <a @click.prevent="handleUpdate">(update)</a>
    </p>`,
```

```
  props: {
    initialCount: {
      type: Number,
      default: 0
    }
  },
  data() {
    return { messageCount: this.initialCount };
  },
  methods: {
    handleUpdate() {
      this.$http.get('/api/new-messages').then((data) => {
        this.messageCount += data.messages.length;
      });
    }
  }
};
```


组件最初会展示从 initialCount 属性传递的通知计数，之后会在每次单击更新链接时发送 API 请求，然后更新计数。

那么该如何为它编写测试呢？有 3 个功能点需要进行测试：

- 应当测试被提供的初始通知计数是否被正确地展示。
- 应当有一个测试来确保当链接被单击时，通知计数被正确地更新。
- 最后，在调用 API 失败时，组件还应当能够优雅地处理异常错误。

现在，先不要在意如何测试有关更新通知计数的部分，仅仅来看初始通知计数。

要测试 NotificationCount 组件，我们可以创建一个新的 Vue 实例，并将 NotificationCount 组件放入其中作为子组件：

```
const vm = new Vue({
  template: '<NotificationCount :initial-count="5" />',
  components: { NotificationCount }
}).$mount();
```

因为没有为 new Vue() 提供 el 属性，所以需要手动调用 .$mount() 方法，以便 Vue 触发挂载进程。否则，将无法对其进行测试。

vm 是 ViewModel 的缩写，它会在本书、vue-test-utils 的文档 和 Vue.js 的文档中被广泛使用。

如果在控制台输出 vm，你会发现它是一个包含众多属性的对象。我们现在需要关心的属性是 $el 属性，它代表由 Vue 生成的组件根元素。

在控制台输出 vm.$el.outerHTML 的结果如下所示：

```
<p>
      Messages: <span class="count">5</span>
      <a>(update)</a></p>
```

可以看到，初始计数已被正确显示。现在来编写一段代码，在组件已被挂载时，如果没有按预期展示数据，则抛出一个错误，这样就不需要每次手动检查 HTML。vm.$el 是一个标准的 DOM 节点，所以可以通过 .querySelector() 方法在它的子元素中查找选择器为 .count 的 span 元素：

```
const count = vm.$el.querySelector('.count').innerHTML;

if (count !== '5') {
  thrown new Error('Expected count to equal "5"');
}
```

如果使用断言库，比如 Chai，并且你也具有编写测试用例的经验，那么第二部分的代码可以简化为你所熟悉的语法：

```
const count = vm.$el.querySelector('.count').innerHTML;
expect(count).toBe('5');
```

现在，如果显示的通知计数不等于 5，测试用例就会抛出一个错误，测试失败。

介绍 vue-test-utils

上一节例子中的语法实际上有些啰唆，而且还只是一个简单的示例。在包含多个事件和模拟组件的复杂示例中，为了测试所编写的组件，可能需要编写比实现组件代码更多的测试代码。

vue-test-utils 是一个协助你编写测试的 Vue 官方库。它提供多种在为组件编写测试时常

用的功能，比如查询 DOM 节点、设置 props 和 data、模拟组件和其他参数、处理事件等。

它不会帮助你运行测试用例（应当使用类似的 Jest 或 Mocha）或者断言（应当使用类似的 Chai 或 Should.js），所以你需要分别安装这些库。在这一章中不会涉及关于测试运行器（test runner）的内容。当测试失败时，所编写的代码仅会抛出错误，同时假设错误也会被捕获。

如同使用其他库那样，可以使用 unpckg 或者别的 CDN 来使用 vue-test-utils，但通常会使用 npm 来安装：

124

```
$ npm install --save-dev vue-test-utils
```

现在，在测试文件中，可以导入 vue-test-utils 并重新编写之前的测试代码，如下所示：

```
import { mount } from 'vue-test-utils'

const wrapper = mount(NotificationCount, {
  propsData: {
    initialCount: 5
  }
});

const count = wrapper.find('.count').text();
expect(count).toBe('5');
```

尽管这段代码的长度也不短 —— 事实上，长度几乎相同 —— 但是它的可读性很好。使用 vue-test-utils 来编写更长、更复杂的例子通常会显著缩小所编写的代码长度。

查询 DOM

在上一节的例子中，你看到了下面的表达式：

```
wrapper.find('.count').text()
```

那么它的作用是什么呢？

首先，从 `mount()` 函数说起。这个函数接受组件和它的一些配置作为参数 —— 在这个例子中，仅包括 props 的数据 —— 并返回一个 *wrapper* 对象，这个对象是由 vue-test-utils 提供的，它将需要进行模拟的组件封装起来以便对它执行不同的操作，并可以在不

使用过多原生浏览器 DOM 方法的情况下查询这些组件。

.find() 是 wrapper 实例提供的方法之一，当提供一个用于查询 wrapper 实例子元素的选择器时，它会返回 DOM 中第一个匹配该选择器的元素节点。从本质上讲，它等价于 DOM 元素节点的 .querySelector() 方法。在 .find() 方法定位到目标元素节点之后，它会返回该元素节点的 wrapper 对象 —— 而不是节点本身。

注意 .find() 仅返回第一个满足选择器规则的元素节点。如果希望得到多个，请使用 .findAll()。它会返回一个 WrapperArray 对象，而不是 Wrapper，但是它具有类似的方法，不过需要使用 .at() 方法来返回 WrapperArray 中指定索引的目标元素节点。

既然我们已经可以通过 wrapper 获取选择器为 .count 的 span 元素节点，就需要另外一个方法来读取它的内容，并测试这些内容等于什么。有两个方法可供使用，分别是 .html() 和 .text()，但是使用 .text() 通常会更好，所以我们会在例子中使用它。

125

现在，已经获取 span 元素节点的内容并存储在一个变量中，之后我们可以使用任何喜欢的方法对它进行测试。

可以使用多种函数来查询 DOM 节点，但值得注意的是，在阅读本书时，这些函数可能已被更改或被添加到浏览器中，所以在这里将它们全部罗列出来是没有意义的，还是去看看文档吧！

挂载选项

如你所见，我们能够为 mount() 提供一个对象作为它的第二个参数，并告诉 Vue 需要让组件拥有什么 props，但还可以通过这个参数来做更多的事。可以通过查看 Vue.js 和 vue-test-utils 的文档了解它的详细用法，在这里仅提供一些最有用的用法：

propsData（正如你之前看到的）是用来传递 props 的：

```
const wrapper = mount(NotificationCount, {
  propsData: {
    initialCount: 5
  }
});
```


slots 是用来传递 components 或者 HTML 字符串的。把第 59 页“具名插槽”章节中的 BlogPost 组件挂载上去：

```
const wrapper = mount(BlogPost, {
  slots: {
    default: BlogContentComponent,
    header: '<h2>Blog post title</h2>'
  },
  propsData: {
    author: blogAuthor
  }
});
```

mocks 是用来为组件实例增加属性的。比如，在这一章的第一个例子中，我们曾使用 this.$http.get() 来发送 HTTP 请求。下面来模拟这个 API：

```
const wrapper = mount(NotificationsCount, {
  mocks: {
    $http: {
      get() {
        return Promise.resolve({ messageCount: 2 });
      }
    }
  },
  propsData: {
    initialCount: 0
  }
});
```

126

现在相比于调用真实的 API（这对于单元测试而言是脆弱而低效的），调用 this.$http.get('/api/new-messages') 将会返回一个立即处于 resolved 状态的 promise。

listeners 是一个包含若干事件监听方法的对象。在大部分情况下，不需要使用这个选项，因为 wrapper 对象的 emitted 属性已经足够，我们将在第 146 页的“测试事件”章节进一步探讨：

```
const wrapper = mount(Counter, {
  listeners: {
    count(clicks) {
      console.log(`Clicked ${clicks} times`);
    }
```


```
  }
});
```

stubs 与 mocks 类似，但不同于对变量进行模拟，它用于对组件进行存根：

```
const wrapper = mount(TheSidebar, {
  stubs: {
    'sidebar-content': FakeSidebarContent
  }
});
```

正如上文提及的，这些只是我所发现的有用选项中的一部分，而更多的选项则会在官方文档中涉及。

模拟和存根数据

上一节简要地介绍了 mocks 选项，它允许你对组件实例添加属性，以模拟那些被增加到 Vue.fn 的方法，使之在 Vue 实例中的任何地方都能访问到。还介绍了 stubs 选项，该选项允许你传入一个假的组件或 HTML 字符串，而不是一个使测试代码产生额外依赖关系的真实组件。这两个选项都通过减少组件间所依赖的可变因素的数量来减少测试的复杂度 —— 如果破坏了一个组件，在理想状况下，你肯定不想让依赖于该组件的所有测试用例也跟着失败吧！

还有4个 .set*() 方法，用于设置 props、data、computed 属性和组件方法。.setComputed() 和 .setMethods() 都是不错的方法，它们可以用来覆盖 computed 属性和组件方法，在模拟稍后会被调用的方法，同时尽可能地简化需要模拟的对象，并由相关的方法在之后的测试中调用（译注：setComputed() 在最新的 API 中已被移除）。

比如，我们可以完全覆盖 handleUpdate 方法，使得消息计数每次递增 2，而不是覆盖 NotificationsCount 的 this.$http.get() 函数：

127

```
const wrapper = mount(NotificationsCount, {
  propsData: {
    initialCount: 0
  }
});

wrapper.setMethods({
  handleUpdate() {
```


```
      this.messageCount += 2;
    }
  });
```

但是做这件事时要格外小心——它可能会使测试变得脆弱——不过如果运用得当，它会成为一个非常有用的功能。

setComputed() 也具有类似的功能，但是需要提供一个特定的值，而不是一个函数：

```
const wrapper = mount(NumberTotal);

wrapper.setComputed({
  numberTotal: 16
});
```

测试事件

最后，vue-test-utils 还包含能够使处理事件变得容易的功能。我认为这是 vue-test-utils 中最有用的部分：如果没有这个库，事件的测试不会变得那么轻松。

首先，来看一看如何在一个组件中触发事件。还记得 NotificationCount 组件中有一个更新消息计数的链接：我们来编写一个测试，该测试会单击链接并校验事件是否被触发。

要测试链接是否被单击，可以通过一个方法模拟 this.$http 的调用过程，之后返回一个 promise（像之前那样），同时还会将一个变量设置为 true 以表示其已经被调用：

```
let clicked = false;

const wrapper = mount(NotificationsCount, {
  mocks: {
    $http: {
      get() {
        clicked = true;
        return Promise.resolve({ messageCount: 1 });
      }
    }
  },
  propsData: {
    initialCount: 2
  }
});
```

现在，当更新操作被触发时，clicked 变量的值应当变为 true。

Wrapper 对象拥有一个 .trigger() 方法，可以使用它来触发单击事件：

```
wrapper.find('a').trigger('click');
```

现在，测试 clicked 是否为 true —— 如果不是，则测试失败：

```
expect(clicked).toBe(true);
```

最后，来看一下如何观察组件触发了哪个自定义事件。每个 wrapper 对象都存储了所有它触发的事件并可以通过 .emitted() 方法获取。这非常有用，意味着你无须添加任何监听器，因为事件都已被捕获！

在第 62 页的“自定义事件”章节，曾看到过一个 Counter 组件，它在被单击时，会触发一个 count 事件，这个事件对象中包含一个它被单击次数的数值，可以写一个测试，来检查这个数值是否准确：

```
const wrapper = mount(Counter);

// 触发 3 次单击事件
wrapper.find('button').trigger('click');
wrapper.find('button').trigger('click');
wrapper.find('button').trigger('click');

const emitted = wrapper.emitted();

expect(emitted.count).toBeTruthy();
expect(emitted.count.length).toBe(3);

expect(emitted.count[0]).toBe(1);
expect(emitted.count[1]).toBe(2);
expect(emitted.count[2]).toBe(3);
```

我们不仅可以观察已发出的事件，还可以观察所有随事件发出的 payload 以及事件发出的顺序。在调试发出事件的组件时，这非常有用。

emittedByOrder() 方法拥有类似的行为，但是它包含一个数组，所以可以观察所有被触发事件的顺序，而不是仅拥有相同名称事件的顺序。

129

总结

在这最后一章中，已经看到如何利用本书到目前为止所有关于组件的知识，并使用 Vue 的官方库 vue-test-utils 来为它们编写单元测试。这个库对 Vue 组件进行封装并提供一些有用的方法，这些方法使你可以对组件、库和属性进行模拟和存根，传递模拟 data 和 props，访问和操纵已生成的元素节点，以及触发和捕获事件。

附录 A

搭建Vue开发环境

每个人都应当从零开始搭建一次大型应用程序的 Vue 开发环境，比如构建工具、测试和所有依赖库。但在实际工作中，这却可能是一种痛苦，因为搭建 Vue 开发环境并不是每次我们都想做的事情。可以通过多种方式来快速搭建一个全功能的 Vue 开发环境，而非亲力亲为，本篇简短的附录将会介绍其中的两种。

vue-cli

vue-cli 使你可以从多个给定模板中，快速引导和搭建 Vue 应用程序。我几乎用它搭建了我的工作涉及的所有 Vue 项目。

要安装 vue-cli，输入：

```
$ npm install --global vue-cli
```

之后，选择想要安装的项目模板，然后运行 npm init [template name]。比如，要想安装 webpack 模板，则运行命令：

```
$ vue init webpack
```

输入命令之后，将会看到一个设置向导，它会提出一系列问题，比如需要安装项目的目录位置、要安装的依赖库以及 npm 的设置信息等。

取决于你选择安装的模板，设置向导看起来将会是这样的：

```
> vue init webpack

? Generate project in current directory? Yes
? Project name vue-test
? Project description A Vue.js project
? Author Callum Macrae <callum@macr.ae>
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? No
? Setup unit tests with Karma + Mocha? No
? Setup e2e tests with Nightwatch? No

   vue-cli · Generated "vue-test".

   To get started:

     npm install
     npm run dev

   Documentation can be found at https://vuejs-templates.github.io/webpack
```

132

当安装了新项目中所有的依赖项之后，就可以运行向导提示给你的启动命令（对于 webpack 模板，就是 npm run dev），然后就可以将搭建好的应用程序投入使用，无须太多配置就能达到预期。

Vue 提供多种可用的官方模板。它们都存储在 GitHub 上的 vuejs-templates 组织中。根据在本文撰写时 GitHub 星标数量的排序，将在下面介绍 6 个可用模板。

webpack

GitHub 中星标数量最多的模板是 webpack，在我看来，这个模板也是最通用的。它提供全功能的 webpack 配置，如使用 vue-loader 来加载单文件组件（*.vue* 文件）、模块热更新、静态代码检测、设置 vue-router、单元测试和功能测试等。

pwa

该模板基于 webpack 模板同时提供它所具备的一切功能，除此之外，它还会将应用程序配置为渐进式 Web 应用——这是一种可离线使用的、能够保存在用户移动设备主屏幕中的 Web 应用。

133

webpack-simple

该模板与 webpack 模板十分相似，但更简单，仅通过 vue-loader 和其他必要 loaders 提

供简单的 wepback 安装配置，去除了模块热更新、静态代码检测以及任何 webpack 模板中所提供的可选功能。

我不推荐使用这个模板。如果你想要一个简单的安装配置，可以使用 webpack 模板并在安装向导过程中，将所有可选的功能设置为 No。该模板并非为生产环境而设计，假如你决定要在多个网站原型上使用这个项目，那么将不得不重复配置 webpack。

simple

simple 模板仅包含一个 HTML 文件和一个 CSS 文件。它非常简单，而且非常适用于不太需要构建系统的场景。

browserify

browserify 模板提供全功能的 Browserify 安装配置。它通过 vueify 来使 Browserify 加载单组件文件、提供模块热更新、静态代码检测和其他可选功能，如 vue-router、单元测试和功能测试等。

browserify-simple

和 webpack-simple 很像，该模板是 browserify 模板的精简版本。

你自己的模板

创建属于你自己的模板也是可行的。如果无法在这些模板中找到适合的模板，或者发现在每次使用其中某个模板时，总是需要做出同样的修改（例如我总是发现自己要将分号添加到 webpack 模板中），那么这可能是最适合你的方式。可以从任何已有模板中拷贝一份或者完全从零开始，vue-cli 的文档介绍了如何能够做到这一点。

Nuxt.js

作为 vue-cli 的可替代方案，可以使用 Nuxt.js 来搭建 Vue 应用。它让你在抽象所有配置的基础上，通过使用 webpack 和 Babel 来配置 Vue、vue-router、vuex 和 vue-meta（用于设置页面标题）创建客户端应用程序。它还提供服务端渲染、自动代码分割、CSS 预处理、伺服静态文件等诸多有用的功能。

附录 B

Vue与React

如果你之前有使用 React 的经验，那么对于 Vue 中的很多相关概念也会很熟悉。我在这个附录中整理了有关两个框架之间异同点的清单，这部分内容基于相当多的示例：通过示例来理解它们之间的异同之处，要比从文字叙述去理解容易得多！

写在前面

Vue 和 React 都拥有相似的工具来搭建简单的应用。React 拥有 create-react-app，通过配置 webpack 来创建 React 应用，从而使你从烦琐的配置文件中摆脱出来。而 Vue 拥有 vue-cli，可以使用它提供的多种模板通过 Webpack 或 Browserify 来定制 Vue，或者不使用任何构建工具直接引用 Vue。如果有定制化的需求，vue-cli 的模板同时也是可配置的，它可以配置 vue-router、单元测试和功能测试（除了 vuex）等配置项，而这些配置项 create-react-app 并未提供。

还可以使用 vue-cli 创建自定义模板。查阅附录 A 以了解更多关于 vue-cli 以及可用模板的内容。

当你需要搭建一个极简配置时，也存在一点儿不同。如果没有构建工具，React 将无法正常工作，因为没有解析器不能编写 JSX 代码，它不被浏览器所支持。所以几乎每个 React 项目都使用了 webpack，而少部分项目使用了 Browserify 。Vue 不需要 JSX 语法（虽然你仍可以使用它），所以无须任何构建工具，就可以在浏览器中直接使用它。大多数项目都不会直接使用，这是因为通常项目中都会有一个模块打包器和一个 ES2015 编译器，不过有这个选项总比没有好。对于初学者来说也是极好的，因为可以在无须关心

任何复杂构建系统的基础上，直接开始编写 Vue。

136

相似性

在 React 和 Vue 之间存在许多相似性。Vue 的设计理念充分汲取了 Angular 和 React 的优点并将它们结合起来，所以如果你真的特别喜欢 React 所拥有的每个特性，那么在 Vue 中同样可以发现它们。这部分内容阐述和演示了一些它们之间的相似点。

组件

React 和 Vue 都是高度组件驱动的。使用组件是一种很好的工作方式：它使你将应用划分为多个合理的、可复用的代码块，这些代码块各自负责应用的不同部分。这也使得它们十分易于测试。React 和 Vue 中的每个组件都拥有自己的状态：在 React 中，它被称作 *state* 并使用 `setState()` 方法进行存储，而在 Vue 中，它叫作 *data*，并通过更改数据对象进行存储。在这两种库中，可以通过几乎相同的方式将数据自上而下地传入组件，并且都被称为 *props*，但语法略微有点儿不同，在本书第 160 页的“差异性”一节将介绍这一点。

通过一个简单的组件演示一下，这个组件接收用户 ID 作为 prop，然后发送一个 API 请求，并将请求所获得的用户数据展示在页面上。

这是 React 中的组件代码：

```
class UserView extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      user: undefined
    };

    fetch(`/api/user/${this.props.userId}`)
      .then((res) => res.json())
      .then((user) => {
        this.setState({ user });
      });
  }, render() {
    const user = this.state.user;
```

```
    return (
      <div>
        {user ? (
          <p>User name: {user.name}</p>
        ) : (
          <p>Loading...</p>
        ) }
      </div>
    );
  }
};

UserView.propTypes = {
  userId: PropTypes.number
};
```


这是 Vue 中的代码：

```
const UserView = {
  template: `
    <div>
      <p v-if="user">User name: {{ user.name }}</p>
      <p v-else>Loading...</p>
    </div>`,
  props: {
    userId: {
      type: Number, required: true
    }
  },
  data: () => ({
    user: undefined
  }),
  mounted() {
    fetch(`/api/user/${this.userId}`)
      .then((res) => res.json())
      .then((user) => {
        this.user = user;
      });
  }
}
```

两个组件的调用方式类似。

在 React 中：

```
<UserGuide userId={10} />
```

在 Vue 中：

```
<UserGuide :userId="10" />
```

React 和 Vue 都具有单向数据流。可以通过 props 将数据传递到组件内部，但是你无法直接修改它。

响应式

React 和 Vue 都拥有相似的响应式机制。响应式是 React 所具备的最好特性之一，所以 Vue 也具备这种特性，真是棒极了！当我们更新 React 中的 state 或 Vue 中的 data 时，或者当某个 prop 又或者某个被库监听的实体发生变化时，依赖于该实体的所有内容均会更新。例如，如果你有一个通过 prop 方式传入另一个组件的 state 并将它渲染在 DOM 中，当这个 state 发生改变时，prop 也将发生改变，内部组件会随之得知 prop 已经发生改变而相应地更新它在 DOM 中的值。

下面来创建一个简单的组件，它在被挂载后从 1 开始进行累加计数，并将结果展示给用户。

这是它在 React 中的组件代码：

```
class TickTock extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      number: 0,
    };
  },
  componentDidMount() {
    this.setInterval(() => {
      this.setState({
        number: this.state.number + 1
      });
```


```
    }, 1000);
  },
  componentWillUnmount() {
    clearInterval(this.counterInterval);
  },
  render() {
    return (
      <p>{this.state.number} seconds have passed</p>);
  }
};
```

这是它在 Vue 中的代码：

```
const TickTock = {
  template: '<p>{{ number }} seconds have passed</p>', data: () => ({
    number: 0,
    counterInterval: undefined,
  }),
  mounted() {
    this.counterInterval = setInterval(() => {
      this.number++;
    }, 1000);
  },
  destroyed() {
    clearInterval(this.counterInterval);
  }
};
```

在这两个例子中，我们在组件的销毁生命周期中添加了一些用于清空 interval 实例的逻辑代码。这是为了防止内存泄漏，同时也将很好地把我们带入下一节的主题：生命周期钩子。

139

生命周期钩子

从 DOM 中创建和移除组件的方式在两个库中是类似的，而且在这两个库中，还可以在组件中增加若干方法以便在组件整个生命周期的各个时间点上运行相关代码。。

创建类钩子

当一个组件被创建并被添加到 DOM 上时调用。在 React 中，拥有如下 4 个钩子函数。

`constructor()`：

在组件挂载之前被调用。

`componentWillMount()`：

在组件即将挂载之前被调用。

`render()`：

在组件挂载过程中被调用，并在这里返回 JSX 模板。

`componentDidMount()`：

当组件挂载后马上被调用。

在 Vue 中，同样拥有 4 个钩子函数，但是它们略有不同。

`beforeCreate()`：

在组件初始化之前被调用。

`created()`：

在组件初始化之后，但是在被添加到 DOM 之前被调用。

`beforeMount()`：

在组件根元素准备好被添加到 DOM 之后被调用。

`mounted()`：

在组件整体挂载时被调用（但是不一定添加到 DOM - 使用 `nextTick()` 来确保根元素已被添加到 DOM）。

更新类钩子

在组件被创建和添加到 DOM 之后，当数据发生改变时，组件将会更新。

140 在 React 中，拥有如下 5 个钩子函数。

`componentWillReceiveProps()`：

在组件 props 更新之前被调用。

`shouldComponentUpdate()`：

在决定组件是否应当被更新时被调用。

`componentWillUpdate()`：

在组件更新之前被调用。

`render()`：

在生成组件应当返回的新的模板标记时被调用。

`componentDidUpdate()`：

在组件更新之后被调用。

在 Vue 中，仅拥有两个钩子函数。

`beforeUpdate()`：

在组件将要被更新之前被调用。

`updated()`：

在组件被更新之后被调用。

在 React 中，如果使用了另一个库来操作 DOM，而这个库与 React 配置并不是十分默契时，通常就会使用 shouldComponentUpdate() —— React 常常会覆盖另一个库的 DOM 修改，除非你告知它别这么做。这个钩子也被用于优化组件渲染性能。在 Vue 中，这通常不是问题，因为它可以和别的库很好地协同工作并自动地进行优化，但是如果要确保 Vue 不会再次更新组件 —— 即使在数据发生改变时 —— 也可以通过添加 `v-once` 属性来告诉 Vue 仅渲染一次。

设置 CSS Class 属性

React 拥有一个广泛使用的库，用于辅助设置 class 属性 —— 叫作 *classnames*。它是这样工作的：

```
<div className={classNames('foo', { bar: isBar })}>...</div>
```

Vue 中也有类似的东西，但是它是内建的。即 v-bind:class 或简写为 :class，它不仅支持表达式，还可以使用对象或者数组：

```
<div :class="['foo', { bar: isBar }]">...</div>
```

141

差异性

React 和 Vue 之间除了拥有很多相似性，还有很多差异性。这里所指的差异性不包含所有细微的差异 —— 语法差异、方法名称差异 —— 仅包含那些在框架基本层面中所体现的差异性。

变更

这两个库中最本质的区别在于，当使用 React 时，直接变更 state 是非常不被鼓励的行为，而在 Vue 中，替换或者修改 data 则是更新它的唯一方式。还可以通过将 Redux 和 vuex 进行比较来看出这一点 —— 在 Redux 中，当你想要修改一个已有的 store 时，会生成一个新的 store，而在 Vuex 中则会直接修改已经存在的 store。在这个附录后面将详细对比 Redux 和 vuex，现在仅需要关注组件的状态。

在 React 中，要更新一个组件的 state，可以使用 setState：

```
this.setState({
  user: {
    ...this.state.user,
    name: newName
  }
})
```

之后，新的 state 将会与当前的 state 对象进行合并（使用浅合并策略）。

而在 Vue 中，则需要直接修改 data：

```
this.user.name = newName;
```

不过这里存在一些注意事项：比如，必须在 user 对象中已经定义了 name 属性，而且还不能用这种方法直接修改数组的元素，你必须使用 splice 来移除旧的元素并将其替换为新的元素。要解决这个问题，可以使用 Vue.set() 方法：

```
Vue.set(this.user, 'name', newName);
```


一般来说，你没有必要担心这一点，但是在遇到状态不更新的问题时，这可能是造成问题的原因。在本书第 15 页“响应式”一节可以找到关于 Vue 中响应式运作方式以及注意事项的内容。

JSX 语法与模板语法

React 和 Vue 之间的另一个本质区别在于页面中数据的渲染方式。在 React 中会使用 JSX，它是由 Facebook 发明、可直接在 Javascript 文件中编写 HTML 的语法。在 Vue 中，如果你愿意，也可以使用 JSX 语法，但大多人还是使用模板语法，它具有类似 Angular 的模板语法、指令和数据绑定语法。

在 React 中，由一个数组生成 HTML 列表的 JSX 代码大概看起来会是这样的：

142

```
render() {
  return (
    <ul>
      {this.state.items.map((item) => (
        <li className={classNames({ active: item.selected })}>
          {item.text}
        </li>))}
      {this.state.items.length ? null : (<li>There are no items</li>
      )}
    </ul>);
}
```

当然，这里可能存在很多种不同的写法。

在 Vue 中，实现同样功能的模板代码如下所示：

```
<ul>
  <li v-for="item in items" :class="{ active: item.selected }">
    {{ item.text }}
  </li>
  <li v-if="!items.length">There are no items</li>
</ul>
```

我比较倾向于模板语法，虽然在使用 React 时，十分喜欢 JSX。如果你真的执着于使用 JSX，也可以在 Vue 中使用它：

```
render(h) {
  return (
```


```
      <ul>
        {this.items.map((item) => (
          <li class={{ active: item.selected }}> {item.text}
          </li>))}
        {this.items.length ? null : (<li>There are no items</li>
        )}
      </ul>);
  }
```

关于如何在应用程序中配置 JSX，请参考第 4 章。

143

CSS Modules

React 和 Vue 中最后一个主要差异点是在 Vue 中编写 CSS 的方式。React 中没有提供相应的内建功能，所以通常会使用 CSS modules。

在 React 中，会像这样使用 CSS modules：

```
.user {
  border: 2px #ccc solid;
  background-color: white;
}

.name {
  font-size: 24px;
}
.bio {
  font-size: 16px;
  color: #333;
}
```

JSX 代码如下：

```
import styles from './styles.css';

    // ...

render() {
  return (
    <div className={styles.user}>
      <h2 className={styles.name}>{user.name}</h2>
      <p className={styles.bio}>{user.bio}</p>
```


```
    </div>);
}
```

Vue 也支持 CSS modules，而且无须配置任何额外插件和构建工具。大概看起来会是这样的（如果使用单文件组件）：

```
<template>
  <div :class="$style.user">
    <h2 :class="$style.name">{{ user.name }}</h2>
    <p :class="$style.bio">{{ user.bio }}</p>
  </div>
</template>

<style module>
  .user { ... }
  .name { ... }
  .bio{...}
</style>
```

但是在大多数情况下，你会注意到人们往往在 Vue 中使用 scoped CSS。在 Vue 中，scoped CSS 的工作原理是为当前组件产生的每个元素生成一个随机的 data-* 属性，然后将其添加至每个元素相应 CSS 选择器的末尾。在 <style> 标签中编写的任何 CSS 代码仅会应用于由该组件生成的元素。

144

使用 scoped CSS 的方式重新编写上一个例子会像这样：

```
<template>
  <div class="user">
    <h2 class="name">{{ user.name }}</h2>
    <p class="bio">{{ user.bio }}</p>
  </div>
</template>

<style scoped>
  .user { ... }
  .name { ... }
  .bio{...}
</style>
```

尽管在 style 标签中的 CSS 代码使用了通用的类名称，而且看起来它们可能会应用于组件外的其他元素，但它们确实将仅适用于该组件元素当中的子元素。话虽如此，我仍然

推荐使用更长、更具有表述性的类名。

scoped CSS 同时也可以和 SCSS 或者其他的 CSS 预处理器搭配和嵌套使用。

生态

由于 React 受欢迎和被广泛使用的程度，它拥有庞大的生态系统。Vue 作为一个较新的、还不太流行的库，目前还没有诸如 React 那样的生态系统和社区 —— 不过它每天都在变得更好，并且当前它的增长速度比 React 的生态系统还要快。

不幸的是，在 React 的生态系统中，尤其是那些至关重要的任务（比如路由）通常可能会非常零散。React 中存在多种路由解决方案 —— 尽管 react-router 似乎是最常用的。而在 Vue 中只有一种 vue-router。虽然 Facebook 已经交由社区去开发那些不属于 React 本身的库，但 Vue、vue-router、vuex、vue-test-utils、vue-cli 以及未来更多的官方库的开发和维护团队却是相同的。

在实际应用中，如果使用的是官方 Vue 插件提供的功能（路由、状态管理、测试），那么你将会拥有不错的开发体验并且无须做出任何选择。但是如果使用的不是 Vue 的官方插件（如国际化插件、HTTP 资源管理插件等），那么你获得的开发体验将远远不如生态更加成熟的 React。

145 现在，来看一看那些能够帮助我们的官方插件。

路由

在 React 中，我们有多种客户端路由解决方案，但到目前为止，使用最广泛的方案仍是 react-router。这个库使用 JSX 语法来描述当页面路径与给定路由规则匹配时所需要显示的组件。

一个使用 react-router 生成的路由如下所示：

```
const app = () => (
  <Router history={hashHistory}>
    <Route path="/" component={PageHome} />
    <Route path="/user/:userId" component={PageUser} />

    <Route path="/settings" component={PageSettings}>
      <Route path="/profile" component={PageSettingsProfile} />
```

```
        <Route path="/email" component={PageSettingsEmail} />
      </Route>
    </Router>
);
```

vue-router 是 Vue 中处理客户端路由的官方库。它使用对象来配置路由而不是 JSX 语法。与上一个例子功能相同的路由如下所示：

```
const router = new VueRouter({
  mode: 'history',
  routes: [
    {
      path: '/',
      component: PageHome
    }, {
      path: '/user/:userId',
      component: PageUser
    },
    {
      path: '/settings',
      component: PageSettings,
      children: [
        {
          path: 'profile',
          component: PageSettingsProfile
        }, {
          path: 'email',
          component: PageSettingsEmail
        }
      ]
    }
  ]
});
```

两个库以不同的方式来完成相同的事情。

146

状态管理

React 中最常用的状态管理库是 Redux，它在 Vue 中等效的官方插件叫作 vuex。它们都使用基本相同的方法，提供一个全局的、可在整个应用程序中存储和修改数据的 store。如果熟悉 Redux，也可以轻松上手 vuex，反之亦然。它们之间的不同点在于前面章节所

讨论的术语和修改的差异性。

使用 Redux 你会拥有一个存储状态（state）的 store。你可以直接访问 state，也可以使用 react-redux 并在中间件中使用 mapStateToProps 函数使其能够作为 prop 被访问。要更新 state，需要使用 reducer 来生成一个新的 state。Reducer 是同步的，要实现异步的话，可以在组件中进行修改，也可以通过插件（如 redux-thunk）来将异步 action 增加到应用程序中。

使用 vuex，同样拥有一个存储状态的 store。可以直接访问 state，但是却无法直接修改它：要更新 state，必须通过 mutation，它是 store 中用于更改数据的特殊方法。Mutation 只能是同步的，所以如果要异步修改某些内容（比如直接在 store 中将一些内容更新为从某个 API 返回的数据），可以使用 action。可以在不使用任何其他插件的情况下完成这些工作 —— 仅包括 Vue 和 vuex。

我们来创建一个用于存储用户列表的 store。

用 Redux 实现如下：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

const initialState = {
  users: undefined
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'SET_USERS':
      return Object.assign({}, state, {
        users: action.payload
      });
    default: return state;
  }
};
const store = createStore(reducer, applyMiddleware(thunk))
```


在另一个单独编写 action 的文件中：

```
export function updateUsers() {
  return (dispatch, getState) => {
```

```
    return fetch('/api/users')
      .then((res) => res.json())
      .then((users) => dispatch({ type: 'SET_USERS', payload: users }));
  };
};
```

使用 vuex 的话，如下所示：

```
const store = new Vuex.Store({
  state: {
    users: undefined
  },
  mutations: {
    setUsers(state, users) {
      state.users = users;
    }
  },
  actions: {
    updateUsers() {
      return fetch('/api/users')
        .then((res) => res.json())
        .then((users) => this.setUsers(users));
    }
  }
});
```

在 React 和 Vue 的组件中，使用 store 的方式完全不同。

在 React 中，你必须将整个组件封装在一个 Redux 中间件中，该中间件使用 mapStateToProps 和 mapDispatchToProps 参数将 action 和 state 以 props 的形式传给组件。

这里演示了在 React 中如何更新 users 并从 store 中获取数据：

```
import { connect } from 'react-redux';
import { updateUsers } from './actions';

class UserList extends React.Component {
  componentDidMount(state, { updateUsers }) {
    updateUsers();
  },
  render({ users }) {
```

```
    return (
      ...
    );
  }
}

148 const mapStateToProps = ({ users }) => users;
const mapDispatchToProps = { updateUsers };

export default connect(mapStateToProps, mapDispatchToProps)(UserList);
```

这个组件在挂载时会获取用户数据，然后当数据加载完成时，users 就能够被使用了，因为 state 被映射到了 props 之上，这是通过 Redux 中间件完成的。

由于 Vue 和 vuex 属于同一个能够协同工作的生态环境，使用起来更加容易。不需要为组件添加任何额外代码，即可以通过 `this.$store` 来访问 data。

以下是在 Vue 中关于上面例子的等效代码：

```
const UserList = {
  template: '...', mounted() {
    this.$store.dispatch('updateUsers');
  },
  computed: {
    users() {
      return this.$store.state.users;
    }
  }
};
```

当组件被挂载时它会获取 users，然后当 users 加载完成时，users 即可通过 `this.users` 来访问。

vuex 也提供一些帮助函数来减少代码的冗余性。想了解更多关于 vuex 的内容，请查阅第 5 章。

组件单元测试

对 React 组件进行单元测试的常用解决方案是使用 Enzyme。*Enzyme* 是一个由 Airbnb 开发的库，使用它可使加载和测试组件变得更加容易。

使用 Enzyme 编写的测试代码看起来会像这样：

```
import { expect } from 'chai';
import { shallow } from 'enzyme';
import UserView from '../components/UserView.js';

const wrapper = shallow(<UserView />);
expect(wrapper.find('p')).to.have.length(1);

const text = wrapper.find('p').text();
expect(text).to.equal('User name: Callum Macrae');
```

Vue 拥有一个非常类似的库 vue-test-utils。与 Enzyme 一样，它也提供加载组件、遍历 DOM 等功能，从而使测试组件变得更加容易。

149

下面是功能类似的测试代码，但它使用 vue-test-utils 来测试 Vue 组件：

```
import { expect } from 'chai';
import { shallow } from 'vue-test-utils';
import UserView from '../components/UserView.vue';

const wrapper = shallow(UserView);
expect(wrapper.contains('p')).toBe(true);

const text = wrapper.find('p').text();
expect(text).toBe('User name: Callum Macrae');
```

正如你所见，它们拥有相似的语法，所以在它们两者之间进行切换将是一件十分容易的事。

索引

Symbols

A

B

C

D

O

P

R

S

T

U

V

W

X

关于作者

Callum Macrae 是一位在英国伦敦就职于 Sam Knows 的 JavaScript 开发工程师。Sam Knows 致力于让所有人都拥有更快的互联网。他热衷于将 Vue 与 SVG 相结合，并定期为开源项目做贡献，包括 gulp 和他自己的一些项目。关于这些，无论是 GitHub 还是 Twitter，都能通过 @callumacrae 发掘到更多。

封面动物

《Vue.js：快跑》的封面动物是欧洲黑鸢（学名 *Milvus migrans migrans*）。

这种夜间猛禽在欧洲中部和东部以及地中海地区随处可见。在温暖的季节里，它们可以远居西部的蒙古国和巴基斯坦；而当冬季降临时，它们就飞往撒哈拉以南的非洲过冬。

黑鸢是鹰科家族中最普遍的一种（同属鹰科的还有鹰、雕、鸢、旧大陆秃鹫和鹞等），在全球约有六百万只，这得益于它们的生活范围、适应能力以及寻找食物的意愿。黑鸢与其他鹰科种类相区别的特征是其头部的白色羽毛、尾部的分叉造型、成一定角度的羽翼以及它独特的叫声——一种尖锐的、时断时续的啸叫声。

O'Reilly 出版的书籍封面上的动物很多已经濒临灭绝，它们对这个世界有着重大的意义。请移步 *animals.oreilly.com* 看看我们能为它们做些什么。

封面的图片来自《英国鸟类》杂志（*British Birds*）。封面所采用的字体是 URW Typewriter 和 Guardian Sans。书中正文所采用的字体是 Adobe Minion Pro，标题采用的字体是 Adobe Myriad Condensed，代码所采用的字体是 Dalton Maag's Ubuntu Mono。

O'REILLY®

Vue.js快跑 构建触手可及的高性能Web应用

带你迅速领略运用Vue.js——组织与简化Web开发中流行的 JavaScript框架——构建既快又灵的单页Web应用。有了这本实践指南，你将迅速完成从基本用法到自定义组件及更多高级特性的领略——甚至包括JavaScript语法扩展JSX。

作者Callum Macrae向你展示了如何使用Vue生态系统中最实用的库，比如实现客户端路由的vue-router和实现状态管理的vuex，以及专门用于测试的vue-test-utils。如果你是一位熟练运用JavaScript、HTML和CSS的前端开发者，那么这本书将向你呈现如何使用Vue来开发一个功能齐全的Web 应用。

- 学会Vue.js基本用法，包括使用模板将数据显示在页面上
- 从零开始或者使用vue-cli从模板创建Vue工程
- 分离代码到独立组件中，从而创建可维护的代码库
- 在Vue.js中使用CSS为网站或网页应用增添样式
- 使用render函数和JSX代替模板语法来定制页面的呈现
- 使用vue-router来操控页面的路由
- 使用vuex来集中进行状态管理
- 使用单元测试来确保组件不出问题

“Callum Macrae 并不想只教会你怎么使用Vue，他还打算告诉你关于构建可伸缩的Vue应用的所有秘诀。显然，他把他的经验都写进书里了。”

——Chris Fritz
Vue核心团队文档负责人

Callum Macrae 是一位在英国伦敦就职于Sam Knows的JavaScript开发工程师。Sam Knows致力于让所有人都拥有更快的互联网。他热衷于将Vue与SVG相结合，并定期为开源项目做贡献，包括gulp和他自己的一些项目。无论是在GitHub上还是在Twitter上，都能通过@callumacrae发掘到更多相关信息。

图书分类：Web开发

责任编辑：张春雨

Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-35299-7

定价：69.00元